UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

# TESINA DE LICENCIATURA

**Título:** **Replicación de bases de datos NoSQL en dispositivos móviles**

**Autor:** Gabriel Capdevila

**Director:** Prof. Mg. Javier Bazzocco

**Carrera:** Licenciatura en Sistemas

## Resumen

*Los dispositivos móviles no quedaron al margen del surgimiento de las bases de datos NoSQL por lo que aumentaron las opciones y las restricciones a tener en cuenta la hora de desarrollar aplicaciones.*

*Este trabajo tiene su eje central en el análisis de bases de datos NoSQL así como en el estudio de distintas técnicas de utilización de dichas bases de datos en ambientes móviles. El objetivo específico es realizar un análisis comparativo de la confiabilidad de una aplicación que utilice una base de datos NoSQL con copias locales y sincronización con un servidor remoto con la de una aplicación que utilice bases de datos enteramente locales en el dispositivo.*

*El caso de estudio a resolver consistirá en el desarrollo de una aplicación que utilice ambos enfoques de estas bases de datos NoSQL, completamente local y con replicación, y en el análisis de la utilización de ésta en diversos dispositivos en un período de tiempo.*

## Palabras Claves

- *NoSQL*
- *Base de datos*
- *Dispositivos móviles*
- *Replicación*

## Conclusiones

*La incidencia de la replicación en la búsqueda es clara. Al momento de hacer una búsqueda la cantidad de veces que se necesitará una conexión a internet fiable es mucho menor en las bases de datos replicadas.*

*Bajo los tests realizados se vio que el tiempo promedio entre fallas para una base de datos es casi tres veces mayor y que el número acumulado de fallas guarda también una relación similar.*

## Trabajos Realizados

- *Desarrollo de una aplicación que utilice dos enfoques para bases de datos NoSQL, una completamente local y otra con replicación.*
- *Evaluación de la cantidad de fallas y del tiempo promedio entre éstas para ambas bases de datos utilizando herramientas de confiabilidad de software.*
- *Comparación los valores para ambas bases de datos.*

## Trabajos Futuros

- *Extender el uso de la aplicación a una mayor cantidad de usuarios para obtener mayor información para los tests.*
- *Realizar estudios y análisis centrados en ambientas más específicos como la utilización de la aplicación bajo una sola opción de red.*
- *Extender el desarrollo de la aplicación a otros usos, para así evaluar los efectos de la replicación en otros ámbitos.*

**Fecha de la presentación:** **JULIO 2015**

Universidad Nacional de La Plata

Facultad de Informática


# Replicación de bases de datos NoSQL en dispositivos móviles

Tesina de Licenciatura en Sistemas

Autor: Gabriel Eduardo Capdevila

Director: Prof. Mg. Javier Bazzocco

JULIO de 2015

# Índice general

# Capítulo 1

# Motivación y objetivos

## 1.1 Motivación

En los últimos años, ha crecido considerablemente la cantidad de dispositivos capaces de conectarse a internet junto con la cantidad de usuarios conectados y las aplicaciones con gran flujo de datos, como las redes sociales y los motores de búsqueda. Con el tiempo, estos servicios y sistemas distribuidos llegaron a tener mucha información como para que su tráfico y cantidad de datos puedan ser manejados por un simple servidor. A su vez, las nuevas arquitecturas distribuidas y modelos de Cloud Computing y Mobile Cloud Computing soportaban el gran flujo de datos alejándose de las arquitecturas de un solo servidor pero generaban otros tantos problemas en los ya existentes sistemas de manejo de bases de datos relacional (RDBMS) cuando se aplican a masivas cantidades de información.

Es entonces cuando comienzan a tomar mayor relevancia las bases de datos NoSQL. Éste es un término genérico utilizado para referirse a una amplia clase de sistemas de almacenamiento de información que no siguen el modelo tradicional RDBMS y no utilizan SQL como lenguaje de consultas. No es un producto ni una tecnología en particular, representa una clase de productos y una colección de diversos y a veces relacionados conceptos sobre almacenamiento y manipulación de datos. NoSQL se utiliza para referirse a las bases de datos que privilegian la escalabilidad y la disponibilidad frente a la atomicidad y la consistencia.

Obviamente, los dispositivos móviles no quedaron al margen de una nueva tecnología que facilitaba la escalabilidad y el uso de modelos distribuidos. Muchas bases de datos NoSQL fueron o creadas exclusivamente para dispositivos móviles o tuvieron su versión ligera adaptada, por lo que

aumentaron las opciones y las restricciones a tener en cuenta la hora de desarrollar aplicaciones móviles.

Una de las primeras restricciones que afrontan este tipo de dispositivos es que la conectividad móvil es altamente variable en performance y confianza. Los dispositivos móviles no pueden garantizar una conexión constante e incluso contando con redes móviles existen problemas de conectividad, habiendo momentos donde la cobertura pueda ser baja o inexistente, especialmente en países en desarrollo. Cuando hablamos del manejo de aplicaciones en móviles y su funcionamiento con Mobile Cloud Computing la conectividad es el problema que surge inherentemente.

Del mismo modo, a pesar de todas las ventajas que presentan las bases de datos NoSQL en cuanto a velocidad y escalabilidad, y por lo que se plantea en " NoSQL in a Mobile World: Benchmarking Embedded Mobile Databases"[1], tener presente bases de datos capaces de manejar NoSQL de manera nativa y offline en un dispositivo móvil no presenta de por sí una ventaja apreciable. En dicha investigación, los autores hacen una comparación de performance de bases de datos SQL, NoSQL y orientadas a objetos en dispositivos móviles. La conclusión a la que llegan es que, si bien se desempeñaba bien en distintas situaciones como búsquedas indexadas y búsquedas simples, para bases de datos locales en móviles NoSQL era superada en performance por bases de datos relaciones

Por estas razones, se debe analizar si es del todo conveniente manejar NoSQL en ambientes móviles con un enfoque completamente nativo u otro completamente orientado a Cloud Computing. Surgen entonces alternativas híbridas entras las completamente nativas y las completamente orientadas a la nube, como el uso de replicación.

## 1.2 Objetivos

Esta propuesta tiene su eje central en el análisis de bases de datos NoSQL así como en el estudio de distintas técnicas de utilización de dichas bases de datos en ambientes móviles.

El objetivo específico es realizar un análisis comparativo de la confiabilidad de una aplicación que utilice una base de datos NoSQL con copias locales y sincronización con un servidor remoto con la de una aplicación que utilice bases de datos enteramente locales en el dispositivo.

Basando esta comparación en la aplicación de Software Reliability Growth Model sobre los distintos tipos y cantidad de fallas que tiene la aplicación en un determinado tiempo en diversos dispositivos, siguiendo las técnicas utilizadas en "Reliability Models Applied to Smartphone Applications" [2].

El caso de estudio a resolver consistirá en el desarrollo de una aplicación que utilice ambos enfoques de estas bases de datos NoSQL, completamente local y con replicación, y en el análisis de la utilización de ésta en diversos dispositivos en un período de tiempo.

Con esto se busca aportar datos de sustento a eventuales desarrolladores que necesiten información para decidir cuál de los diversos enfoques sería más conveniente para sus proyectos.

# Capítulo 2

# Los desafíos de los RDMBS e introducción a NoSQL

## 2.1 Los retos de los RDBMS

Hoy en día es claro que los sistemas de manejo de bases de datos relacional (RDBMS) tienen sus propios problemas cuando se aplican a masivas cantidades de información. Los problemas se refieren a procesamiento eficiente, exponencial crecimiento de la cantidad de usuarios e información, escalabilidad, paralelización efectiva y costos [3].

### 2.1.1 BigUsers

Hace no tanto tiempo, 1.000 usuarios diarios en una aplicación era mucho y 10.000 era un caso extremo. Hoy, la mayoría de las nuevas aplicaciones son almacenadas en la nube y disponibles en internet, donde soportan usuarios globales 24 horas, 365 días al año. Más de 2.000 millones de personas están conectadas mundialmente a internet, y el tiempo que pasan online está creciendo sostenidamente, creando una explosión en el número de usuarios concurrentes. Hoy en día, no es raro que una aplicación tenga millones de usuarios en un día [4].

Manejar un gran número de usuarios concurrentes es importante, pero, ya que los requerimientos de uso de la aplicación son difíciles de predecir, es igual de importante manejar rápida y dinámicamente el creciente o decreciente número de usuarios concurrentes:

- Una aplicación recién lanzada puede tornarse viral, creciendo de cero a millones de usuarios en una noche.
- Algunos usuarios están activos frecuentemente mientras otros usan la aplicación unas veces, para nunca volver.
- La promoción o el lanzamiento de nuevos productos pueden aumentar dramáticamente el uso de la aplicación.

El gran número de usuarios combinados con la naturaleza dinámica del uso de patrones lleva a la necesidad de tecnologías de bases de datos que sean fácilmente escalables. Con tecnologías relacionales, muchos desarrolladores de aplicaciones encuentran difícil, o incluso imposible, obtener la escalabilidad dinámica y el nivel de escala que necesitan manteniendo el performance que los usuarios demandan.

### 2.1.2 BigData

Lo que se considera BigData fue variando con el tiempo y lo seguirá haciendo. Hoy en día, cualquier cantidad de información que supere algunos terabytes es clasificada como Big Data. Éste es típicamente el tamaño donde el data set es suficientemente grande como para expandirse a distintas unidades de almacenamiento. Es también el tamaño sobre el cual las técnicas RDBMS tradicionales empiezan a mostrar los primeros signos de complicaciones [3].

Incluso hace un par de años, un terabyte de información personal puede haber parecido un poco exagerado, aunque hoy en día sea común adquirir discos de este tamaño. En los próximos años, no sería sorprendente si los discos rígidos por defecto en las computadoras tengan más que unos pocos terabytes. Se vive una era de un constante crecimiento de la información. Consumimos un montón de información y producimos otra tanta.

Es difícil medir el verdadero tamaño de la información digitalizada o el tamaño de la Internet. Igualmente, algunos estudios estiman que el tamaño es inmensamente grande y en el rango de los zettabyte (1.000.000.000 de terabytes) o mayor aún. En un estudio titulado "The Digital Universe Decade – Are you ready? " [5] se presenta una vista en el corriente estado de la información digital y su crecimiento. El estudio dice que el tamaño total de la información digital creada y replicada crecerá a 35 zettabytes para 2020. El estudio también afirma que la cantidad de información producida y disponible ahora está superando la cantidad de almacenamiento disponible.

Mientras crece el tamaño de la información y las fuentes de creación de esta información se diversifican, los siguientes retos se intensificarán:

Almacenar y acceder eficientemente grandes cantidades de información es difícil. Las adicionales demandas de tolerancia a fallos y producción de backups harán las cosas más complicadas.

Manipular grandes data sets involucrará correr inmensos procesos paralelos. Recuperarse adecuadamente de alguna falla durante dichos procesamientos y proveer resultados en un período de tiempo razonablemente corto es complejo.

Manejar los continuamente evolutivos esquemas y metadata para data semi-estructurada y no-estructurada, generada por diversas fuentes.

Por lo tanto, las formas de almacenar y recuperar grandes cantidades de información necesitan nuevos métodos. NoSQL y soluciones relacionadas a BigData son de los primeros pasos en esa dirección.

### 2.1.3 Escalabilidad

La escalabilidad es la habilidad de un sistema de incrementar el rendimiento agregando recursos para direccionar los incrementos de carga. Ésta puede ser lograda proveyendo grandes recursos que logren satisfacer las nuevas demandas o bien puede ser lograda apoyándose en un cluster de máquinas ordinarias que trabajen como una unidad [3].

El involucramiento de máquinas grandes y poderosas es típicamente clasificado como escalabilidad vertical. Proveer supercomputadoras con muchos núcleos de CPU y grandes cantidades de almacenamiento directamente ligado es una típica solución de escalabilidad vertical. Dichas opciones de escalamiento vertical son por lo general caras y propietarias.

La alternativa a la escalabilidad vertical es la escalabilidad horizontal. La escalabilidad horizontal involucra un cluster de sistemas commodity donde el cluster aumenta según aumente la carga. La escalabilidad horizontal típicamente involucra agregar nodos adicionales para procesar cargas adicionales.

La llegada de BigData y la necesidad de procesamiento paralelo a gran escala para manipular esta información han llevado a la adopción general de infraestructuras escalables horizontalmente. Algunas de estas estructuras escaladas horizontalmente en Google, Amazon, Facebook, eBay y Yahoo! involucran un gran número de servidores. Algunas de estas infraestructuras tienen miles e incluso cientos de miles de servidores.

### 2.1.4 Bases de datos relacionales

Los retos de los RDBMS sobre procesamiento de datos masivos en la web no son específicos de un producto, sino que pertenecen a todas las clases de dichas bases de datos. RDBMS asume una bien definida estructura de datos y que dichos datos son uniformes. A su vez, trabaja sobre el prerrequisito de que las propiedades de los datos pueden ser definidas de antemano y que sus interrelaciones están bien establecidas y son referenciadas sistemáticamente. También asume que los índices pueden ser definidos consistentemente en datasets y que dichos índices pueden ser uniformemente analizados para una búsqueda más rápida [3].

Desafortunadamente, RDBMS empieza a mostrar signos de ceder terreno cuando estas asunciones no se cumplen. RDBMS puede ocuparse ciertamente de algunas irregularidades y una falta de estructura pero en el contexto de datasets escasos con estructuras definidas vagamente, RDBMS parece una solución forzada.

Con datasets masivos los típicos mecanismos de almacenamiento y métodos de acceso también se ven insuficientes. Denormalizar las tablas, quitar restricciones y aminorar las garantías transaccionales pueden ayudar a escalar el RDBMS, pero luego de estas modificaciones éste comienza a parecerse a un producto NoSQL.

La flexibilidad tiene un precio. NoSQL alivia los problemas que RDBMS tiene y hace que sea fácil trabajar con grandes cantidades de data, pero a cambio quita el poder de la integridad transaccional y la indexación y búsqueda flexible.

## 2.2 Introducción a NoSQL

### 2.2.1 Contexto e historia

Las bases de datos no relacionales no son nuevas. De hecho, los primeros stores no relacionales se remontan a cuando fueron inventadas las primeras máquinas computacionales. Las bases de datos no relacionales progresaron a través de la aparición de los mainframes y existen hace años en ambientes especializados y específicos [3].

Igualmente, los stores no relacionales que aparecieron en el mundo NoSQL son una nueva encarnación que nació en el mundo de las aplicaciones de internet escalables masivamente. Estos stores NoSQL no relacionales fueron concebidos, en su gran mayoría, en el mundo de la computación distribuida y paralela.

A través de los años, Google ha construido una infraestructura escalable masivamente para su buscador y sus aplicaciones. Su forma de abordar el problema era resolviendo cada nivel de la pila de aplicación, construyendo una infraestructura escalable para procesamiento paralelo de grandes cantidades de información. Entonces Google creó BigTable, un mecanismo completo que incluía un filesystem distribuido, un datastore column-family, un sistema de coordinación distribuido y un ambiente de ejecución paralelo basado en MapReduce. Luego Google presentaría y publicaría una serie de papers explicando algunas de las piezas claves de la infraestructura.

MapReduce es un framework para procesar grandes conjuntos de datos distribuidos a través de diferentes máquinas. La idea principal detrás de MapReduce es mapear el dataset en una colección de pares clave-valor para luego reducir todos los pares con la misma clave en uno solo [6] **(Figura 1)**.

La publicación de los papers de Google levantó mucho interés en los desarrolladores open-source. Los creadores del motor de búsqueda open-source, Lucene, fueron los primeros en desarrollar una versión libre que replicara algunas de las características de la infraestructura de Google. Subsecuentemente, se unieron a Yahoo, donde crearon un universo paralelo que imitaba todas las partes del sistema de Google al que llamaron Hadoop. En algún momento durante la creación de Hadoop fue cuando surgió la idea de lo que sería NoSQL, y la emergencia de éste sentó las bases para su rápido crecimiento. Es importante considerar que el éxito de Google ayudó a propulsar

una saludable adopción de la nueva era de conceptos de computación distribuida, el proyecto Hadoop y NoSQL.

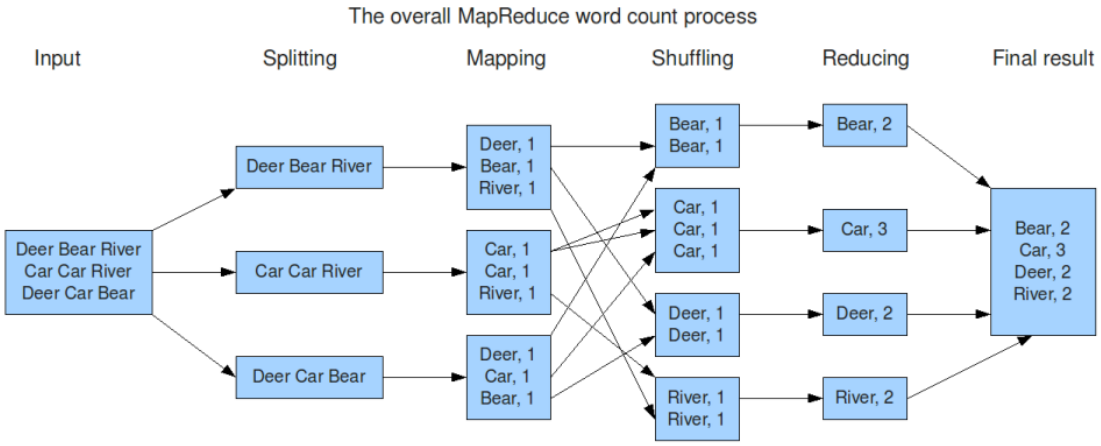


*Figura 1 Diagrama Map Reduce*

Además, un año luego de que la publicación de los papers de Google haya captado interés en el procesamiento paralelo escalable y datastores distribuidos, Amazon decidió compartir parte de su trabajo. En el 2007, Amazon presentó sus ideas de un datastore distribuido de gran disponibilidad y eventualmente consistente llamado Dynamo.

Con el apoyo hacia estas tecnologías de parte de dos gigantes de la industria, Google y Amazon, surgieron muchos productos nuevos en este espacio. Muchos desarrolladores comenzaron a jugar con la idea de usar estos métodos en sus aplicaciones y muchas empresas, desde startups a grandes corporaciones, comenzaron a abrirse a aprender esta nueva tecnología. En menos de 5 años, NoSQL y conceptos relacionados para manejar Big Data se hicieron populares y emergieron distintos casos de uso por parte de compañías bien conocidas como Facebook, Netflix, Yahoo, Ebay y muchas más. Muchas de estas compañías incluso han contribuido al hacer open source sus extensiones y productos.

El término NoSQL primero hizo su aparición a fines de los 90s como el nombre de una base de datos relacional open source creada por Carlo Strozzi. Esta base de datos almacena sus tablas en archivos ASCII representando cada tupla por una línea con los campos separados por tabs. El nombre viene del hecho de que la base de datos no usa SQL como lenguaje de consulta. En cambio, la base de datos es manipulada a través de shell scripts que pueden ser combinados en pipelines de UNIX. Además de haber coincidencias en los términos, el NoSQL de Strozzi no tuvo influencia alguna en las bases de datos que hoy se conocen como NoSQL [7, CAPÍTULO 2].

El término "NoSQL" como lo conocemos hoy en día puede rastrearse a una meetup del 11 de junio de 2009 en San Francisco organizada por Johan Oskarsson, un desarrollador viviendo en Londres. El ejemplo de BigTable y Dynamo había inspirado nuevos proyectos que experimentaban con

almacenamientos de datos alternativos, y las discusiones sobre éstos se habían convertido en una parte importante de las conferencias de software más relevantes. Oskarsson estaba interesado en aprender más sobre algunas de estas nuevas bases de datos mientras se encontraba en un summit de Haddop en San Francisco, pero, como no tenía demasiado tiempo como para visitar todas las alternativas, decidió organizar una meetup donde pudieran juntarse todos y mostrar su trabajo a quién estuviera interesado.

Johan necesitaba un nombre para la meetup, algo corto, recordable y sin demasiadas búsquedas en Google que pueda convertirse en un buen hashtag de Twitter y sea fácil de encontrar en una búsqueda. Pidiendo sugerencias en el chat de Cassandra, se quedó con la de "NoSQL" de Eric Evans que, a pesar de tener la desventaja de ser un término negado y no ser realmente descriptivo de los nuevos sistemas, sí juntaba los requisitos para el hashtag. En ese momento sólo pensaban en ponerle nombre a un simple meetup y ciertamente no esperaban que ese nombre se convierta en el nombre la nueva tecnología en su totalidad.

El término "NoSQL" tomó rápido reconocimiento, pero nunca ha sido un término que haya tenido una definición clara. El llamado original del "NoSQLMeetup" era para "bases de datos no relacionales, distribuidas y open source" y las charlas fueron sobre Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB y MongoDB, pero el término nunca se ha reducido a este grupo en particular. No hay una definición aceptada generalmente ni una autoridad para proveer una. Todo lo que se puede hacer es discutir algunas características comunes de las bases de datos que tienden a llamarse NoSQL.

### 2.2.2 ¿Qué es NoSQL?

Teniendo en cuenta cuáles son los mayores retos de las bases de datos RDBMS y las características que buscan cumplir, se puede examinar qué sería NoSQL.

Para comenzar, hay un punto obvio que es que las bases de datos NoSQL no usan SQL. Algunas tienen lenguajes de consultas, e incluso algunas han implementado lenguajes de consultas similares a SQL para que sean más fáciles de aprender. Pero hasta ahora ninguna ha implementado algo que se acerque mínimamente al estándar SQL [7, CAPÍTULO 2].

Otra característica importante de estas bases de datos es que por lo general son proyectos open source. Aunque el término NoSQL es frecuentemente utilizado a sistemas propietarios, hay una noción de que NoSQL es un fenómeno open source.

Muchas bases de datos NoSQL han sido desarrolladas por la necesidad de correr en clusters, especialmente aquellas discutidas en el meetup original. Esto tiene un efecto en el modelado de datos y en el enfoque de consistencia. Las bases de datos relacionales usan transacciones ACID para manejar la consistencia en toda la base de datos. Esto es, transacciones que cumplen un mínimo de cuatro características esenciales: atomicidad, consistencia, aislamiento y durabilidad (por sus siglas en inglés: Atomicity, Consistency, Isolation y Durability). Dichas características garantizan la ejecución correcta y completa de las mismas transacciones, así como la consistencia

y el funcionamiento correcto de la base de datos a través del tiempo. Esto choca inherentemente con un ambiente distribuido en un cluster, por lo que las bases de datos NoSQL ofrecen un rango nuevo de opciones para la consistencia y distribución.

De todas formas, no todas las bases de datos NoSQL están fuertemente orientadas a correr en clusters. Existe una clase de estas bases de datos, las bases de datos basadas en Grafos, que usan un modelo de distribución similar al de las bases de datos relacionales pero ofrecen un modelo de datos diferente para manejar relaciones complejas.

Así mismo, las bases de datos NoSQL operan sin esquema, permitiendo agregar campos a registros sin tener que definir los cambios previamente en la estructura. Esto es particularmente útil cuando se maneja información no uniforme con datos variables para cada registro.

Todas estas características mencionadas son comunes a lo que describimos como bases de datos NoSQL, pero así mismo, ninguna de éstas hacen a la definición de NoSQL, y es probable que nunca exista una definición coherente de ésta. Como se ha dicho, NoSQL ha abierto el rango de opciones para el almacenamiento de datos y este nuevo paradigma no debe ser confinado a lo que se cataloga generalmente como NoSQL. Aun así, este conjunto de características se ha convertido en una suerte de guía para lo que representa NoSQL.

Mucha gente considera que el término "NoSQL" significa realmente "NotOnly SQL" (*No sólamente SQL*). Pero, además de la mínima diferencia de escritura de que en este caso debería ser "NOSQL" y no "NoSQL", no tendría mucho sentido el término ya que en este nuevo significado también podrían encasillarse Oracle y Postgres, mezclando un poco los términos.

La interpretación "*not-only*" tiene un sentido si se describe el paradigma de lo que mucha gente piensa que es el futuro de las bases de datos. Esta es de hecho una contribución bastante acertada, ya que es mejor pensar en NoSQL como un movimiento y no como una tecnología en particular. Las bases de datos no sólo no dejarán de usarse sino que probablemente seguirán siendo las más comunes y usadas.

La diferencia radica en que ahora se ven a las bases de datos relacionales como una opción más para el almacenamiento de datos. Este punto de vista es referido frecuentemente como *polyglot persistance* (algo así como persistencia poliglota) donde se usan diferentes data stores en diferentes circunstancias. En vez de simplemente elegir una base de datos relacional porque todo el mundo lo hace, se debe entender la naturaleza de la información que estamos almacenando y cómo queremos manipularla. El resultado es que muchas organizaciones tendrán una mezcla de diferentes tecnologías de almacenamiento de datos para diferentes circunstancias.

### 2.2.3 ¿Por qué usar NoSQL?

Big Users, Big Data y la Escalabilidad están cambiando la manera en la que están siendo desarrolladas muchas aplicaciones. La industria ha estado dominada por bases de datos relacionales por 40 años, pero los desarrolladores se están pasando cada vez más a bases de datos

NoSQL principalmente por tres razones: mejor productividad de desarrollo a través de modelos de datos más flexibles; mayor posibilidad de escalar dinámicamente para manejar más usuarios y más datos; y, performance mejorada para satisfacer las expectativas de los usuarios que necesitan aplicaciones altamente responsivas y para manejar procesamientos de datos más complejos [4].

En cuanto al modelado de los datos, las bases de datos relacionales y NoSQL se manejan de maneras muy diferentes. El modelo relacional toma la información y la separa en muchas tablas interrelacionadas, organizándola en filas y columnas. A su vez, las tablas se relacionan entre ellas a través de claves foráneas. El modelo relacional minimiza la cantidad de almacenamiento requerido, ya que cada porción de información es almacenada en un solo lugar, un requerimiento clave cuando las bases de datos relacionales fueron creadas y el almacenamiento en disco era escaso.

Sin embargo, la eficiencia del espacio trae consigo un incremente en la complejidad de la búsqueda de datos. La información deseada debe ser recolectada de muchas tablas y combinadas antes de proveerlas a la aplicación. De manera similar, cuando se escribe la información, la escritura debe ser coordinada y realizada en muchas tablas.

Las bases de datos NoSQL tienen modelos muy diferentes. Por ejemplo, una base de datos NoSQL orientada a documentos toma la información y la agrupa en documentos JSON y cada uno de estos documentos JSON puede ser analizado como un objeto para ser utilizado por la aplicación. Uno de estos documentos podría, por ejemplo, tomar toda la información almacenada en una fila que abarque 10 tablas de una base de datos relacional y agruparla en un solo documento. Manejar la información agrupada de este modo podría llevar a duplicar dicha información. Pero, ya que el almacenamiento no es más prohibitivo en cuánto a costos, la flexibilidad resultante, la fácil distribución de los documentos y las mejoras en la performance de lectura/escritura hacen que sea una concesión interesante para aplicaciones basadas en la web.

Otra gran diferencia es que las tecnologías relacionales tienen esquemas rígidos mientras que los modelos NoSQL son libres de éstos. Las tecnologías relacionales requieren una estricta definición del esquema antes de almacenar cualquier tipo de datos en la base. Cambiar el esquema una vez que los datos son almacenados suele ser algo complejo. Cambiar el funcionamiento, almacenar nueva información, etc, suele generar cambios que rompen la aplicación y que son frecuentemente evitados. Exactamente todo lo contrario al comportamiento deseado en la era de Big Data, donde los desarrolladores de aplicaciones necesitan constante y rápidamente incorporar nuevos tipos de información para enriquecer sus aplicaciones.

En comparación, las bases de datos basadas en documentos son libres de esquema, permitiendo añadir libremente nuevos campos a los documentos JSON sin tener que definir esos cambios de antemano. El formato de la información que se almacena puede cambiar en cualquier momento, sin interrumpir el funcionamiento de la aplicación. Esto permite a los desarrolladores reaccionar rápidamente para incorporar nueva información a sus aplicaciones.

Por lo que se refiere a lidiar con el aumento de usuarios concurrentes y cantidad de información, las aplicaciones y sus bases de datos subyacentes se deben escalar de una o dos maneras: utilizando scale up (escalabilidad vertical) o scale out (escalabilidad horizontal). Mientras que scale up implica un enfoque centralizado que se base en servidores cada vez más grandes, scale out implica un enfoque distribuido que organiza muchos servidores estándares, físicos o virtuales.

Antes de que existan las bases de datos NoSQL, el enfoque predeterminado para la escalabilidad de bases de datos era la ténica scale up. Esto se basaba en la arquitectura fundamentalmente centralizada de las tecnologías de bases de datos relacionales. Para soportar más usuarios concurrentes y/o almacenar más información, se necesitaban servidores cada vez más y más grandes con más CPUs, más memoria y más capacidad de disco para el almacenamiento de todas las tablas. Los servidores grandes tienden a ser altamente complejos, propietarios and desproporcionadamente caros, al contrario del hardware simple y económico típicamente utilizado en la capa de aplicación.

Con la tecnología relacional, mejorar un servidor es un ejercicio que requiere planificación, adquisición y tiempo de inactividad para completarse. Dado el relativamente impredecible ritmo de crecimiento de usuarios en las aplicaciones actuales, es difícil evitar un abastecimiento erróneo de recursos. Con demasiados recursos se puede gastar de más y con pocos recursos los usuarios pueden experimentar una mala experiencia o la aplicación en sí puede fallar. Y, ya que todos los recursos están en el mismo servidor, se debe manejar una excelente estrategia de tolerancia a fallos y alta disponibilidad.

En cambio, las bases de datos NoSQL fueron desarrolladas desde un principio para ser bases de datos distribuidas con un enfoque scale out. Utilizan un cluster de servidores estándares, físicos o virtuales para almacenar la información y dar soporte a las operaciones propias de la base de datos. Para escalar, se suman servidores adicionales al cluster y la información y las operaciones se distribuyen a lo largo del cluster ahora más grande. Ya que se espera que los servidores comunes que conforman estos clusters puedan fallar de vez en cuando, las bases de datos NoSQL están construidas para tolerar fallos y recuperarse de ellos, haciéndolas altamente flexibles.

Las bases de datos NoSQL proveen un enfoque linear mucho más simple para escalar bases de datos. Si se agregan diez mil nuevos usuarios a la aplicación, se puede simplemente agregar otro servidor al cluster, y así sucesivamente. No hay necesidad de modificar la aplicación mientras se va escalando ya que la misma aplicación siempre ve una base de datos única y distribuida.

Por lo general, un enfoque es escalamiento scale out distribuido también termina siendo más barato que la alternativa scale up. Esto se debe a que los servidores grandes, complejos y tolerantes a fallas son difíciles de diseñar, construir y mantener. Además, mientras los costos de licencia de bases de datos relacionales comerciales pueden ser prohibitivos, las bases de datos NoSQL por lo general son open source y relativamente baratas.

Todo lo visto hasta ahora no significa necesariamente que sólo sea conveniente utilizar bases de datos NoSQL cuando se comiencen a tener problemas de escalabilidad o se desee hacer una

representación de datos libre sin esquemas. Las bases de datos NoSQL tienen mucho que ofrecer más allá que solucionar los problemas de escalabilidad y el modelado de datos ya mencionados.

Otra de las utilidades es que puede mejorar el tiempo de desarrollo de la aplicación. En gran medida gracias a que no se tiene que lidiar con complejas queries SQL, al poder evitar hacer JOINs entre todas las bases finales de la aplicación.

También se ven ampliamente beneficiados por la velocidad de ejecución. Por más que se tenga una pequeña cantidad de información, si se puede responder a las búsquedas en milisegundos en vez de cientos de milisegundos, se tiene una mayor probabilidad de ganarse el visto bueno de los usuarios. Especialmente en aplicaciones móviles o de dispositivos que no estén continuamente conectados.

Un motivo igualmente importante es evitar la siempre presente frustración de la impedancia incorrecta (*impedance mismatch)* al realizar el mapeo entre bases de datos relacionales y lenguajes de programación, comúnmente orientados a objetos. Hasta ahora se ha analizado el impacto de las bases de datos NoSQL poniendo el foco en Big Data sobre clusters, que si bien se considera un factor clave que llevo a reevaluar el mundo de las bases de datos, no es el único motivo por el cual cada vez más equipos consideran las bases de datos NoSQL.

Se ha creado una oportunidad para repensar las necesidades de almacenamiento y muchos equipos de desarrollo ven que usando bases de datos NoSQL se puede aumentar la productividad simplificando el acceso a las bases de datos aunque no se tenga necesidad de escalar más allá de una sola máquina.

Los desarrolladores comúnmente usan lenguajes de programación orientados a objetos para construir aplicaciones y generalmente es más eficiente trabajar con información que se encuentra en la forma de un objeto con una estructura compleja compuesta de datos anidados, listas, arreglos, etc. El modelo de datos relacional provee una estructura de datos muy limitada que no se mapea necesariamente bien con un modelo de objetos. En cambio, la información debe ser almacenada y recuperada de decenas de tablas interrelacionadas. Existen frameworks que proveen algún alivio en la interacción entre objetos y bases de datos relacionales pero la *impedance mismatch* fundamental aún existe entre la forma en la que una aplicación quisiera ver la información y la forma en la que es efectivamente almacenada en la base de datos relacional.

Por otro lado y para dar un ejemplo, las bases de datos orientadas a documentos, pueden almacenar un objeto entero en un simple documento JSON y soportan estructuras de datos complejas. Esto hace que sea más fácil conceptualizar la información así como escribir y desarrollar aplicaciones, incluso a veces con menos líneas de código.

En conclusión, todos estos motivos podrían resumirse en dos razones primarias para considerar NoSQL. La primera es manejar el acceso a la información con tamaños y performance que demandan un cluster; y la otra es mejorar la productividad del desarrollo de una aplicación utilizando un estilo de interacción de datos más conveniente.

# Capítulo 3

# Tipos y características de NoSQL

## 3.1 Tipos de bases de datos NoSQL

Desde un principio, las bases de datos NoSQL se dividieron en cuatro tipos distintos: clave-valor, basadas en documentos, tabulares o column-family y basadas en grafos.

Una base de datos clave-valor es una simple tabla hashing, usada principalmente cuando todo el acceso a la base de datos es a través de una clave primaria. Se podría pensar como una tabla en un modelo RDBMS tradicional con dos columnas, ID y NOMBRE, siendo la columna del ID la clave y NOMBRE el campo que almacena el valor. Las bases clave-valor son las más simples de usar dentro de las diferentes clases de NoSQL disponibles. El cliente puede o bien obtener el valor a través de su clave, poner un valor para cierta clave o borrar una clave de la base de datos. El valor es un blob (Binary Large OBjects) que la base de datos solo almacena, sin importarle o saber qué hay dentro. Es responsabilidad de la aplicación entender que ha sido almacenado. Ya que las bases clave-valor siempre se acceden por clave principal, generalmente tienen una gran performance y pueden ser fácilmente escalables [7, CAPÍTULOS 8, 10 y 11].

El modelo orientado a columnas, *column-family*, permite que la información sea guardada más efectivamente, ya que evita usar espacio para almacenar nulls simplemente al no almacenar una columna cuando un valor no existe para dicha columna. El BigTable de Google contiene un modelo donde la información es guardada en columnas, contrastando con el modelo RDBMS que guarda la información en filas. Cada unidad de información puede ser pensada como un conjunto de pares clave/valor, donde la unidad en sí es identificada con la ayuda de un identificador principal, por lo general llamado clave principal. BigTable y sus derivados tienden a llamar a esta clave principal la

row-key. A su vez, almacenadas de una forma ordenada, las unidades de información son organizadas y ordenadas basándose en la row-key [3].

Las bases de datos basadas en grafos permiten almacenar entidades y relaciones entre estas entidades. Las entidades también son conocidas como nodos, los cuales tienen propiedades y vendrían a ser una instancia de un objeto en la aplicación. Las relaciones son conocidas como aristas que pueden tener propiedades. Las aristas tienen un significado direccional y los nodos son organizados por las relaciones que permiten hallar patrones entre éstos. La organización del grafo permite que la información sea almacenada una vez y luego interpretada de diferentes maneras basándose en las relaciones.

La investigación de este trabajo se perfilará hacia las bases de datos basadas en documentos, por lo que si bien se hace una breve descripción de los cuatro tipos de bases de datos NoSQL, se le otorgará, obviamente, más importancia al tipo de base de datos que más nos interesa.

## 3.2 Bases de datos basadas en documentos

Obviamente, los documentos son el concepto principal en las bases de datos NoSQL basadas en documentos. La base de datos almacena y recupera documentos, que pueden ser XML, JSON, BSON o muchos otros. Estos documentos son estructuras de datos jerárquicas auto describibles que pueden contener mapeos, colecciones o valores escalares. Los documentos almacenados son similares unos a otros pero no tienen que ser en sí exactamente iguales [7, CAPÍTULO 9].

Las bases de datos basadas en documentos almacenan documentos en la parte del valor del store clave/valor. Sería algo similar a una base de datos clave/valor donde el valor podría ser examinable.

Estas bases también permiten que el esquema de datos pueda diferir entre los documentos y aún así éstos puedan ser parte de la misma colección, a diferencia de RDBMS donde cada fila dentro de una tabla tiene que tener el mismo esquema. Por ejemplo, se tienen los siguientes documentos:

```
{
"firstname": "Martin",
"likes": [ "Biking", "Photography" ],
 "lastcity": "Boston",
"lastVisited":
}
```

```
{
"firstname": "Pramod",
"citiesvisited": [ "Chicago", "London", "Pune", "Bangalore" ],
"addresses": [   { "state": "AK",     "city": "DILLINGHAM",     "type": "R"   },
                 { "state": "MH",     "city": "PUNE",     "type": "R" } ],
"lastcity": "Chicago"
}
```

Viendo los documentos, se pueden ver que son similares, pero tienen diferencias en el nombre de los atributos. Justamente, esto es de lo permitido en bases de datos basadas en documentos.

Se representa una lista de ciudades visitadas (cities visited) como un array o una lista de direcciones (addresses) como una lista de documentos embebida dentro del documento principal. Incluir documentos como sub-objetos dentro de otros documentos provee un mejor acceso y una mejor performance.

Si se ven los documentos, se podrá apreciar que algunos de estos atributos son similares, como el nombre (firstname) o la ciudad (city). Al mismo tiempo, hay atributos en el segundo documento que no existen en el primero, como direcciones (addresses), y otros tanto que sí están en el primero y no en el segundo, como gustos (likes).

Esta forma diferente de representar la información no es igual a RDBMS donde cada columna tiene que ser definida, y si no tiene datos es marcada como vacía o seteada en null. En los documentos, no hay atributos vacíos. Si un atributo en particular no se encuentra, se asume que no fue seteado o no es relevante para el documento. Los documentos permiten que se creen nuevos atributos sin la necesidad de definirlos previamente o cambiar los documentos ya existentes.

Algunas de las bases de datos basadas en documentos más populares son MongoDB, CouchDB, Terrastore, OrientDB, RavenDB y Lotus Notes.

### 3.2.1 Características

Si bien existen muchas bases de datos basadas en documentos especializadas, se usará MongoDB para ejemplificar su funcionamiento. Siempre teniendo en cuenta que cada producto puede tener ciertas características que podrían no encontrarse en otras bases de datos [7, CAPÍTULO 9].

Cada instancia de MongoDB tiene múltiples bases de datos y cada base de datos puede tener múltiples colecciones. Cuando se compara esto con RDBMS, una instancia RDBMS es lo mismo que una instancia MongoDB, los esquemas en RDBMS son similares a las bases de datos MongoDB, y las tablas RDBMS son colecciones en MongoDB. Cuando almacenamos un documento, se debe

elegir a qué base de datos y a qué colección pertenece. Por ejemplo, *database.collection.insert(document)*, el cuál por lo general es representado como *db.coll.insert(document)*.

#### 3.2.1.1 Consistencia

La consistencia en MongoDB es configurada usando el método de réplica y la opción de esperar a que todos los writes sean replicados a todos o a un número específico de esclavos. Cada escritura puede especificar el número de servidores a los que tiene que ser propagada antes de considerarse exitosa.

Un comando como *db.runCommand({ getlasterror : 1 , w : "majority" })* le dice a la base de datos que tan fuerte es la consistencia que se quiere. Por ejemplo, si se tiene un servidor y se especifica *w* como *majority*, la escritura retornará inmediatamente ya que hay solamente un nodo. Si se tienen tres nodos en el set de réplica y se especifica *w* como *majority*, la escritura tendrá que completar un mínimo de dos nodos antes de reportarse como exitosa.

Se puede incrementar el valor *w* para una mayor consistencia pero se sufrirá en el performance, ya que ahora las escrituras se tendrán que ejecutar en más módulos.

El método de réplica también permite incrementar la performance de lectura al permitir la lectura desde los esclavos seteando el parámetro slaveOk. Este parámetro puede ser seteado en la conexión, en la base de datos, en una colección o individualmente para cada operación.

De forma similar a varias opciones de lectura, se puede cambiar la configuración para lograr mayor consistencia de escritura. Por defecto, una escritura es considerada exitosa una vez que la base de datos la recibe. Esta configuración se puede cambiar para esperar a que la escritura se haya sincronizado con el disco o se haya propagado a dos o más esclavos. Esto se conoce como WriteConcern. Uno se asegura que ciertas escrituras sean escritas al maestro y algunos esclavos seteando WriteConcern en REPLICAS_SAFE.

Hay que lograr un equilibrio, basándose en las necesidades de la aplicación y los requerimientos del negocio, al decidir que configuraciones pueden ser más útiles para slaveOK durante la lectura y qué nivel de seguridad se desea utilizando WriteConcern durante la escritura.

#### 3.2.1.2 Transacciones

Las transacciones, en el sentido RDBMS tradicional, permiten comenzar a modificar la base de datos con comandos insert, update o delete sobre diferentes tablas y luego decidir si conservar los cambios utilizando commits o rollbacks. Estas construcciones generalmente no están disponibles en soluciones NoSQL, donde un write simplemente tiene éxito o falla.

Las transacciones a nivel de documentos simples son llamadas transacciones atómicas, mientras que las transacciones que involucren más de una operación no son posibles, aunque existan productos como RavenDB que sí soporten transacciones a través de múltiples operaciones.

Por defecto, todas las escrituras son reportadas como exitosas, aunque se puede igualmente realizar un control más fino usando el parámetro WriteConcern. Se puede asegurar que una orden sea escrita a más de un nodo antes de reportarla como exitosa usando WriteConcern.REPLICAS_SAFE. Los diferentes niveles de WriteConcern permiten elegir los niveles de seguridad durantes las escrituras. Por ejemplo, al escribir entradas de un log, se puede setear el nivel de seguridad más bajo, WriteConcern.NONE.

#### 3.2.1.3 Disponibilidad

Las bases de datos basadas en documentos intentan mejorar la disponibilidad replicando data usando la configuración maestro-esclavo. La misma información está disponible en múltiples nodos y los clientes pueden acceder a la información incluso cuando el nodo principal está caído. MongoDB implementa la replicación y provee una muy alta disponibilidad usando los conjuntos replicados.

En los conjuntos replicados, hay dos o más nodos participando en una réplica maestro-esclavo asincrónica. Los nodos del conjunto de réplica eligen al maestro entre ellos mismos. Asumiendo que todos los nodos tienen iguales derechos en la votación, algunos pueden ser favorecidos por estar más cerca a otros servidores, por tener más RAM, etc. Los usuarios a su vez pueden influir al asignar un número de prioridad entre 0 y 100 a un nodo.

Todos los requerimientos van hacia el nodo maestro, y la información es replicada a los nodos esclavos. Si el maestro se cae, los nodos restantes en el conjunto replicados votan entre ellos para elegir al nuevo maestro. Todos los requerimientos futuros son luego redirigidos al nuevo maestro, y los nodos esclavo comienzan a obtener información del nuevo maestro. Cuando el nodo que falló vuelve a estar disponible, se une como un esclavo y se pone al tanto con el resto de los nodos al conseguir toda la información que le falta.

La **figura 2** es un ejemplo de configuración de un conjunto de replicados. Se tienen dos nodos, mongo A y mongo B, corriendo la base de datos MongoDB en el data center principal y mongo C en el data center secundario. Si se quieren que los nodos en el data center principal sean electos como nodos primarios, se les puede asignar una prioridad mayor que a los otros nodos. Se pueden también agregar a los conjuntos sin tener que poner offline a los que están.

La aplicación lee o escribe del nodo maestro. Cuando la conexión es establecida, la aplicación solo necesita conectarse con un nodo (sea el primario o no) en el conjunto replicado y el resto de los nodos son descubiertos automáticamente. Cuando el nodo maestro se cae, el driver se comunica con el nuevo nodo primario elegido por el conjunto de réplica. La aplicación no tiene que manejar nada de los fallos de comunicación o el criterio de selección de nodos. Usar el conjunto de replicados permite tener una alta disponibilidad en el datastore.

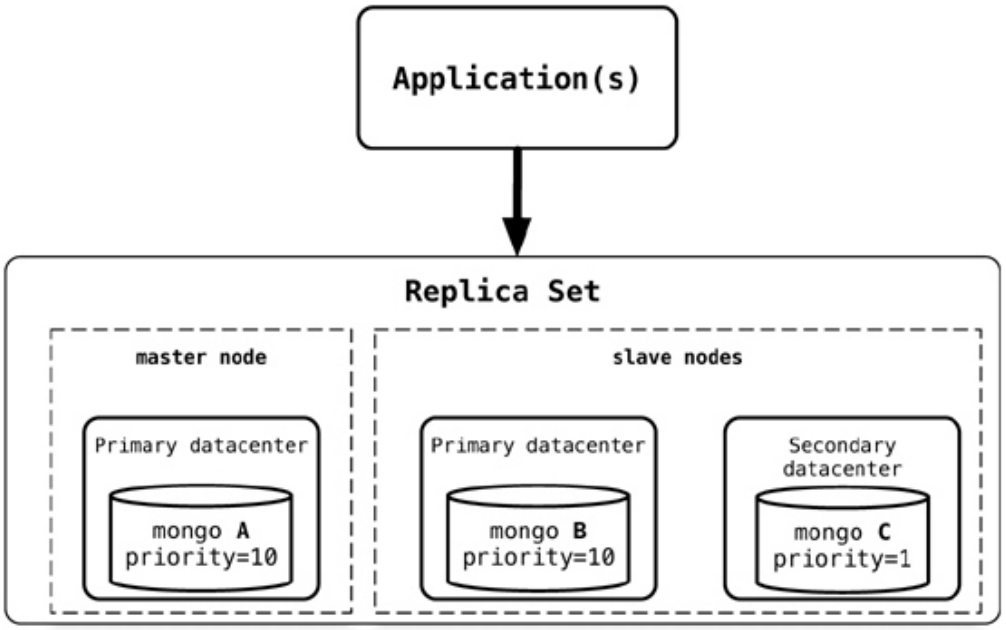


*Figura 2 Diagrama de un conjunto de replicados*

Los conjuntos de replicados son generalmente usados para redundancia de datos, recuperación de fallos automática, escalabilidad de lectura, mantenimiento de servidores sin darlos de baja y recuperación de desastres. Se pueden lograr configuraciones de disponibilidad similares con CouchDB, RavenDB, Terrastore y otros productos.

#### 3.2.1.4 Características de las queries

Las bases de datos basadas en documentos proveen tipos de queries diferentes. CouchDB permite la utilización de queries a través de vistas, queries complejos sobre documentos que pueden o bien ser materializados (Materialized Views) o dinámicos (como las vistas RDBMS que pueden ser materializados o no). Con CouchDB si se necesitara sumar el número de reviews de un producto y además obtener el rating promedio, se podría agregar una vista via MapReduce para devolver el número de reviews y el valor promedio de sus ratings.

Cuando hay muchos requerimientos, no es ideal computar la suma y el promedio para cada uno de ellos. En cambio, se puede agregar una vista materializada que compute previamente los valores y almacene los resultados en la base de datos. Estas vistas materializadas son actualizadas cuando hace una búsqueda, si es que se ha cambiado algún dato desde la última actualización.

Una de las mejores características de las bases de datos basadas en documentos, comparándolas con las bases de datos clave/valor, es que se puede buscar información dentro del documento sin tener que devolver todo el documento por su clave y luego buscar dentro de él. Esta característica posiciona a este tipo de base de datos más cerca del sistema de queries del model RDBMS.

MongoDB tiene un lenguaje query expresado a través de JSON con constructores como $query para la cláusula *where,* $orderby para ordenar la información o $explain para mostrar el plan de ejecución de la búsqueda. Existen muchos otros constructores como éstos que pueden ser combinados para crear una query de MongoDB.

Por ejemplo, analizando qué queries se pueden lograr con MongoDB, suponiendo que se quiera retornar todos los documentos en una ordered collection pertenecientes a un *customerId*883c2c5b4e5b, el SQL sería:

*SELECT * FROM order WHERE customerId = "883c2c5b4e5b"*

*El query equivalente en MongoDB en cambio sería:*

*db.order.find({"customerId":"883c2c5b4e5b"})*

De manera similar, al elegir *orderId* and *orderDate* de un cliente:

*db.order.find({customerId:"883c2c5b4e5b"},{orderId:1,orderDate:1})*

Los queriescount, sum y demás también están disponibles del mismo modo. Ya que los documentos son objetos agregados, es muy fácil buscar documentos que tengan que coincidir usando campos con objetos hijos. Suponiendo que se quiera buscar todas las órdenes donde uno de los ítems ordenados tenga un nombre como *Refactoring*. El SQL sería:

*SELECT * FROM customerOrder, orderItem, product*

*WHERE customerOrder.orderId = orderItem.customerOrderId*

*AND orderItem.productId = product.productId*

*AND product.name LIKE '%Refactoring%'*

Y el equivalente en Mongo sería:

*db.orders.find({"items.product.name":/Refactoring/})*

Los queries de MongoDB son más simples ya que los objetos se encuentran embebidos dentro de un documento simple y se puede buscar basándose en los documentos hijos embebidos.

**3.2.1.5 Escalabilidad**

La idea de escalar es la de añadir nodos o cambiar el almacenamiento sin tener que migrar la base de datos a un recipiente más grande. No se habla acerca de hacer cambios en la aplicación para que soporte más carga, sino, se habla acerca de qué características posee la base de datos que le permita manejar más carga.

Escalar para soportar más carga de lectura puede lograrse añadiendo más esclavos de lectura, para que todas las lecturas puedan ser redirigidas a éstos. Dada una aplicación con una alta carga de lectura, con el cluster con conjunto de réplica de tres nodos, se puede añadir mayor capacidad de lectura al mismo según sea necesario agregando más nodos esclavos con el flagslaveOK. Esto se

llama escalado horizontal para lectura. Una vez el nuevo nodo, mongo D en este caso, es iniciado, necesita ser agregado al conjunto de réplica:

*rs.add("mongod:27017");*

Cuando un nuevo nodo es añadido, se sincronizará con los nodos existentes, se unirá al conjunto de réplica como un nodo secundario y comenzará a responder requerimientos de lectura. Una ventaja de esta configuración es que no se tienen que no se tienen que reiniciar a ninguno de los otros nodos y que no hay tiempo muerto para la aplicación.

Cuando se quiere escalar para la escritura se puede hacer *sharding* con la información. El sharding es similar a las particiones en RDBMS donde se divide la data por valor en una cierta columna, como por ejemplo *provincia/estado* o *año*.

Con RDBMS, las particiones usualmente se tratan del mismo modo, para que la aplicación cliente no tenga que buscar en una partición específica sino que puede buscar en la tabla base; el RDBMS se encarga de encontrar la partición correcta para la query y devuelve los datos.

Cuando se aplica sharding la información también es dividida según ciertos campos, pero luego es movida a diversos nodos Mongo. La data es movida dinámicamente entre los nodos para asegurarse que los *shards* están siempre balanceados.

Se pueden añadir más nodos al cluster y aumentar el número de nodos escribibles habilitando el escalado horizontal para escrituras:

*db.runCommand( { shardcollection : "ecommerce.customer*

*key : {firstname : 1} } )*

Dividir la información en el nombre (*firstname*) del cliente asegura que la data esté balanceada a través de los *shards* para una performance de escritura óptima. Lo que es más, cada *shard* puede ser un conjunto de réplica asegurando una mejor performance de lectura entre *shards*.

Cuando se añade un nuevo *shard* a este cluster ya existente, la información será balanceada a través de la nueva cantidad de nodos. Mientras sucede todo este movimiento de información y refactorización de infraestructura la aplicación no experimenta ningún tipo de downtime, aunque puede que el cluster no tenga un performance óptimo cuando grandes cantidades de información son movidas para re-balancear los shards **(Figura 3)**.

El key del shard juega un rol muy importante. Ya que lo ideal sería ubicar los shards de la base de datos MongoDB cerca de los usuarios, hacer sharding basado en ubicación puede ser una buena idea. Por ejemplo, en EEUU, cuando se hace sharding basado en ubicación toda la información de usuarios de la costa este se encuentra en los *shards* que son servidores de dicha costa y toda la información de usuarios de la costa oeste se encuentra en los *shards* pertenecientes a la costa oeste.

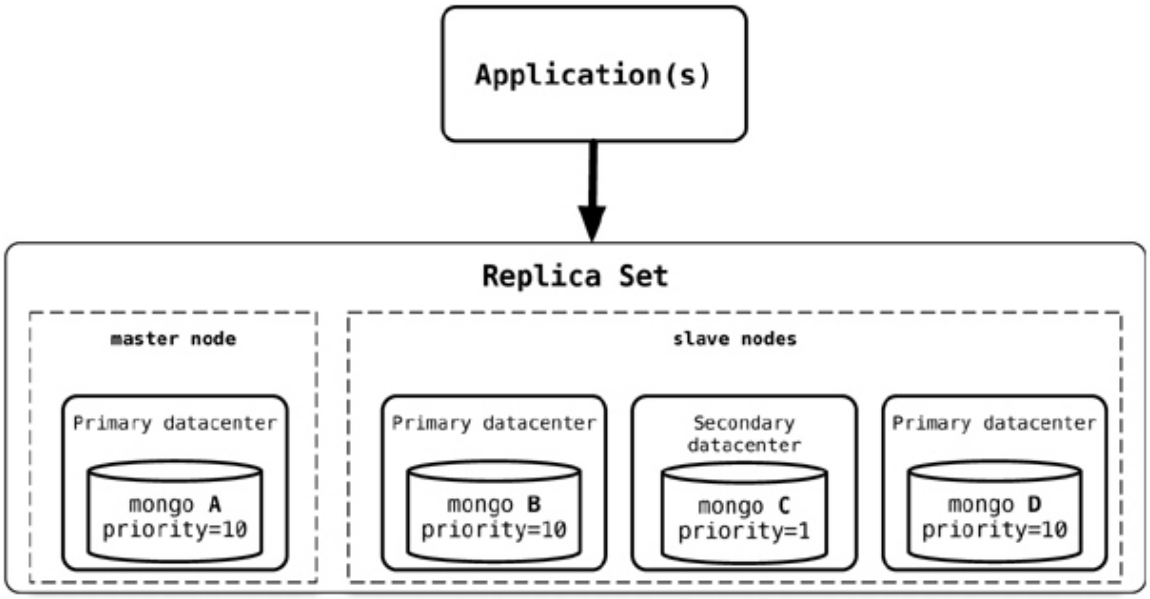


*Figura 3 Ejemplo Escalabilidad*

### 3.2.2 Casos en los que conviene y no conviene usar

Algunas de las situaciones en las cuáles las bases de datos basadas en documentos son útiles:

- *Logs de eventos*:
  Las aplicaciones tienen diversas necesidades en cuanto al loggeo de eventos. Dentro de una empresa existen muchas aplicaciones diferentes que quieren hacer un log de eventos. Las bases de datos orientadas a documentos pueden almacenar todos estos diferentes tipos de eventos y pueden actuar como una central de datos para el almacenamiento de eventos. Es sirve específicamente cuando el tipo de dato que va a ser almacenado por los eventos cambia continuamente. A los eventos se le pueden aplicar sharding por el nombre de la aplicación que originó el evento o el tipo de evento al que pertenezca.

- *Plataformas CMS y de blogging*:
  Ya que las bases de datos basadas en documentos no tienen un esquema predefinido y usualmente comprenden documentos JSON, se adaptan bien a sistemas CMS o aplicación de publicación online, manejar comentarios de usuarios, registro de usuarios, perfiles y documentos web.

- *Analíticas web o de tiempo real*:
  Estas bases de datos pueden almacenar analíticas en tiempo real. Ya que parte del documento puede ser actualizado, es muy fácil almacenar vistas de páginas o visitores únicos. Y además se pueden añadir nuevas métricas sin cambios en el esquema.

- *Aplicaciones E-commerce*:
  Las aplicaciones E-commerce por lo general necesitan tener un esquema flexible para productos y órdenes, así también como la habilidad de evolucionar sus modelos de datos sin un complejo refactoring de la base de datos o migración de datos.

Por otro lado, existen situaciones en las cuáles no conviene en ningún sentido utilizar bases de datos basadas en documentos:

- *Transacciones complejas que involucren diversas operaciones*:
  Si se necesita tener operaciones atómicas entre documentos, entonces puede fallar. De todas formas, existen algunas bases de datos basadas en documentos que soportan este tipo de operaciones, como RavenDB

- *Queries sobre estructuras agregadas variantes*:
  Esquemas flexibles significan que la base de datos no opone ninguna restricción a los mismos y que la información se guarda en la forma de entidades de aplicaciones. Si se necesita hacer un query ad hoc sobre estas entidades, las mismas queries cambiarán de una a otra. En términos RDBMS significa que mientras se hagan *joins* sobre las tablas, estas tablas a unir no serán iguales entre sí. Ya que la data se guarda como un agregado, si el diseño de éste cambia constantemente, se necesita guardar estos agregados al nivel más bajo de granularidad. Básicamente, se debe normalizar la información. En este escenario, las bases de datos basadas en documentos seguramente no funcionarán.

## 3.3 Modelos de distribución en NoSQL

Uno de los principios claves de las bases de datos NoSQL es su habilidad de ejecutarse en un cluster de servidores en vez de una sola máquina, al contrario de las bases de datos relacionales tradicionales, las cuales están diseñadas para ser centralizadas en un solo servidor. Esto permite escalar el sistema cuando la información aumenta y se necesitan más recursos. El uso que pretendemos darle a nuestro sistema determinará qué modelo es mejor para nuestro caso en particular [8, CAPÍTULO 3].

Hay principalmente dos técnicas de las cuales derivan muchos modelos de distribución: sharding y replicación. Esencialmente, el sharding divide la información a través de diferentes nodos, mientras que la replicación duplica la información en varios nodos.

### 3.3.1 Único servidor

El enfoque más simple es no tener ningún tipo de distribución alguna, ejecutando la base de datos y almacenando la información en un único servidor. Esta podría ser la elección más adecuada para

muchas aplicaciones que no requieren ningún tipo de distribución, evitando agregar complejidad innecesaria al sistema ya que todas las solicitudes de lectura y escritura son manejadas por la misma entidad.

### 3.3.2 Replicación maestro-esclavo

Esta replicación duplica la información a través de varios nodos, uno siendo el maestro y otros tantos los esclavos. El nodo maestro es el responsable de almacenar la información, mientras que las lecturas pueden hacerse tanto del maestro o los esclavos. Para asegurarse que los nodos esclavos vean la última información disponible en todo momento, un proceso de replicación sincroniza la información del maestro a los esclavos. Este modelo es especialmente beneficioso cuando la información está sujeta a una mayoría de operaciones de lectura.

Hay dos ventajas principales para este tipo de distribución. Por un lado, es muy fácil escalar el modelo agregando nuevos nodos esclavos que ayuden a reducir la carga de los ya existentes. Por otro lado, los nodos esclavos pueden continuar manejando lecturas incluso si el nodo maestro falla. Por lo que aunque fallen las escrituras el alcance de la falla es reducido efectivamente y puede ser recuperado convirtiendo cualquier esclavo en el nuevo maestro.

De todas formas, hay algunas cuestiones que hacen este modelo un enfoque inconveniente, especialmente si el sistema maneja muchas solicitudes de escritura. El nodo maestro puede convertirse en el cuello de botella del modelo ya que solo éste puede manejar las solicitudes de escritura. Incluso si falla el nodo maestro, todas las escrituras no persistidas se perderán. Marcar un nodo esclavo como el nuevo nodo maestro solucionará muchos problemas, pero para entonces algunas operaciones podrían ya haberse perdido.

Por último, el proceso de sincronización tiene una gran desventaja, la inconsistencia. Si bien éste es un problema inherente a las bases de datos NoSQL, en el caso de la replicación maestro-esclavo diferentes nodos esclavos podrían ver diferente información en un punto dado del tiempo si el nodo maestro no ha propagado aún los cambios.

### 3.3.3 Sharding

El sharding divide la información en muchas partes y le asigna a cada una un shard, un servidor único que procesará todas las solicitudes de escritura y lectura asociadas a la información que contiene.

Idealmente, cada shard deberá procesar aproximadamente la misma cantidad de solicitudes por lo que la carga estará bien balanceada entre todos los servidores. Esto es muy difícil de lograr ya que la forma en la que la información es accedida depende de muchos factores, como la localización geográfica o la hora del día, por ejemplo. Algunas bases de datos NoSQL presentan sharding automático, tomando la responsabilidad de propagar la información a muchos servidores automáticamente.

Mientras que la replicación presenta una mejora en el acceso a la lectura de datos, el sharding puede mejorar sustancialmente la performance general ya que se ocupa tanto de escrituras como lecturas, haciéndola una mejor opción si el sistema debe manejar muchas operaciones de escritura.

Por otro lado, sharding no es mucho mejor que un sistema con un único servidor en cuanto a la tolerancia de fallos. Si uno de los muchos servidores falla, la porción de la información almacenada en este se torna inaccesible.

### 3.3.4 Peer-to-peer

Peer-to-peer es el modelo de distribución más complejo y pretende solucionar una de las desventajas de los modelos de replicación maestro-esclavo, la habilidad de manejar sistemas con una gran carga de escrituras. La idea principal es eliminar el rol del nodo maestro, esencialmente creando un número de servidores replicados capaces de procesar tanto solicitudes de lectura como de escritura.

Todos los nodos tienen, teóricamente, exactamente la misma cantidad de datos, por lo que una falla en cualquiera de ellos no previene el acceso a la información. Es también muy fácil agregar nuevos nodos al modelo para mejorar la performance.

De todas formas, la replicación peer-to-peer también presenta complicaciones. Al igual que la replicación maestro-esclavo, la inconsistencia es un gran problema. El hecho de que cada nodo sea capaz de manejar solicitudes de escritura significa que eventualmente la misma información pueda ser actualizada al mismo tiempo en dos nodos diferentes, generando conflictos.

### 3.3.5 Combinaciones de sharding y replicación

No sólo es posible utilizar cualquiera de las estrategias ya mencionadas sino que también se pueden combinar para generar modelos más complejos. Combinar replicación maestro-esclavo con sharding genera múltiples nodos maestros, pero sin información duplicada a través de ellos ya que cada porción de información solo tiene un maestro.

Además, combinar replicación peer-to-peer con sharding es posible. En este caso, la información es compartida a varios nodos, pero no todos los nodos tienen la misma información.

## 3.4 Modelado de datos en agregados

Un modelo de datos es el modelo a través del cual percibimos o manipulamos nuestros datos. El modelo de datos describe cómo interactuamos con los datos en la Base de Datos. Es distinto al modelo de almacenamiento, el cuál describe cómo la BD almacena y manipula la información internamente. En un sistema ideal, deberíamos poder ignorar el modelo de almacenamiento, pero

en la práctica necesitamos al menos algún conocimiento de éste, principalmente para lograr una mejor performance [7, CAPÍTULO 2].

El modelo relacional toma la información que queremos almacenar y la divide en tuplas. Una tupla es una estructura de datos limitada que captura un conjunto de valores. No se puede anidar una tupla dentro de otra para obtener tuplas anidadas ni se puede incluir una lista de valores o tuplas dentro de otras. La simplicidad es la clave del modelo relacional. Nos permite pensar que todas las operaciones devuelven y trabajan sobre tuplas.

En cambio, cada tipo de base de datos NoSQL usa un modelo de datos distintos, los cuáles se pueden separar en cuatro categorías: **clave-valor**, **documental**, **tabular** y **basada en grafos**. De éstas, las primeras tres comparten una característica en común en sus modelos de datos que es una orientación a formar agregados.

La orientación a formar agregados toma una aproximación distinta. Reconoce que de vez en cuando, se puede querer operar sobre información organizada en unidades que tienen una estructura más compleja que un par de tuplas. Puede ser útil pensar en términos de un registro complejo que permita listas y otras estructuras anidadas dentro de él. Las bases de datos **clave-valor**, **documental** y **tabular** hacen uso de este registro de datos más complejo.

No existe un término común para este tipo de registros complejos, por lo que se usará el término agregado proveniente del término "*aggregate*" en inglés.

### 3.4.1 Diferencias agregados entre relacional y NoSQL

Se puede utilizar un sistema de ventas online para mostrar un ejemplo que marque las diferencias entre estos agregados y un modelo de datos relacional. Supongamos que de este sistema de ventas se quiera almacenar información sobre usuarios, el catálogo de productos, órdenes de compra, direcciones de envío e información de pago.

Para modelarlo usando un modelo de datos relacional tradicionalmente usaríamos una tabla para modelar cada una de estas entidades donde cada una esté propiamente normalizada para que no haya información repetida entre las tablas y además tendríamos integridad referencial **(Figura 4)**.

Hablando ahora en términos más orientados a agregados, podríamos representar este modelo con dos agregados: clientes y órdenes. El cliente contiene una lista de direcciones de facturación, la orden contiene una lista de items pedidos, una dirección de envío e información de pago. El pago por sí mismo contiene una dirección de facturación por ese pago **(Figura 5)**.

Un registro de una dirección lógica aparece tres veces en la información del ejemplo, pero en vez de usar identificadores es tratado como un valor simple y copiado cada vez que es necesario.

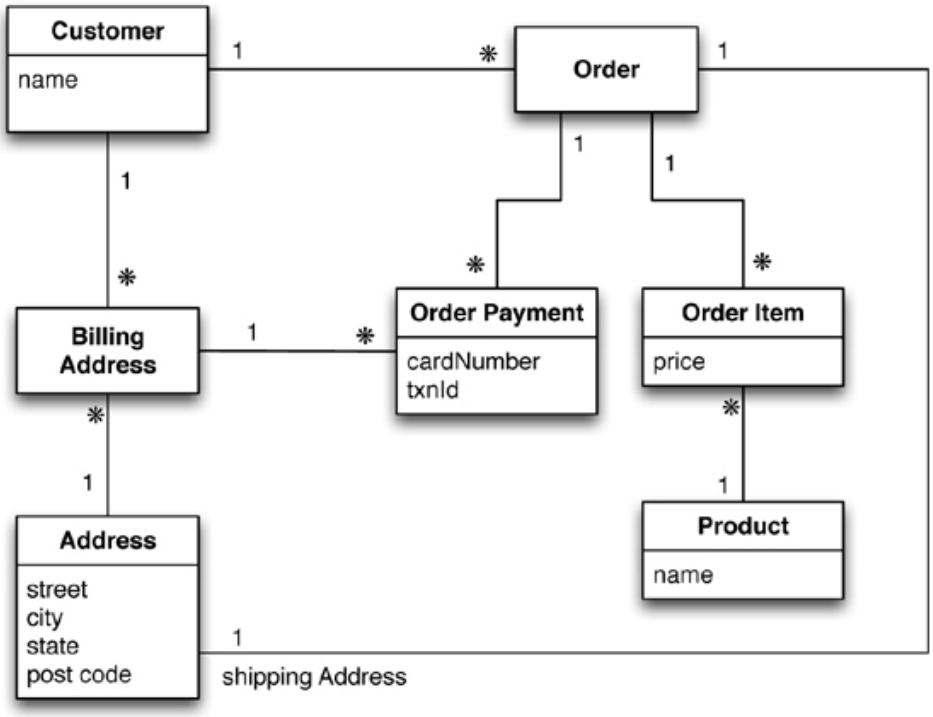


*Figura 4 Modelado usando un modelo de datos relacional*

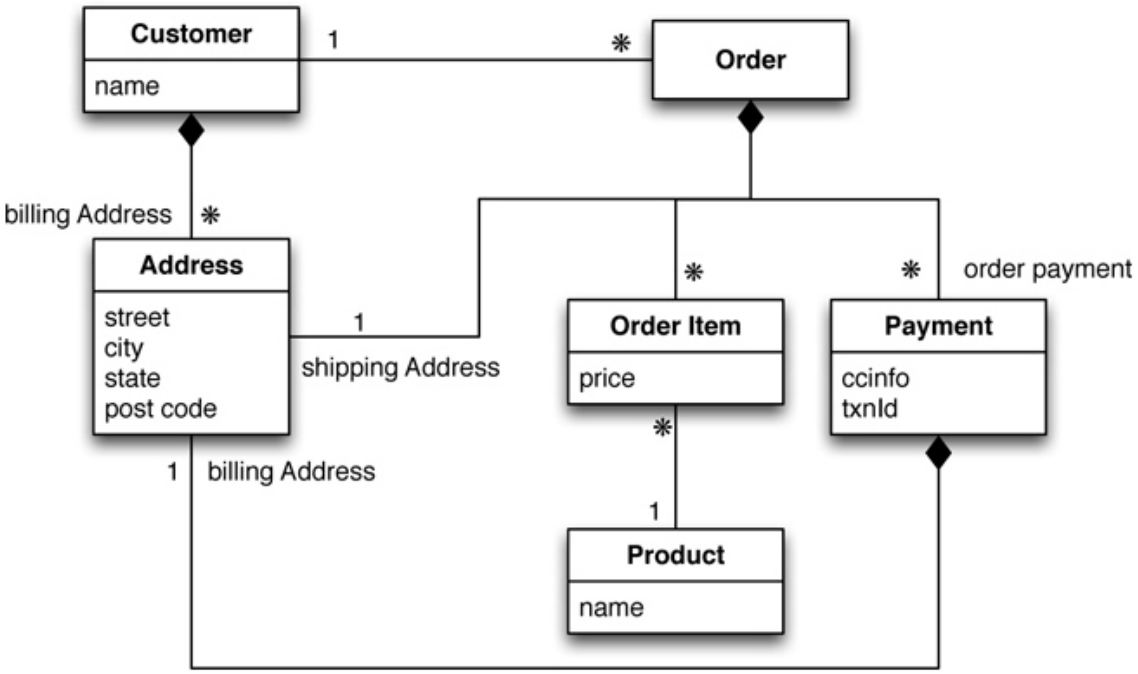


*Figura 5 Modelado usando un modelo de datos agregado*

El link entre el cliente y la orden no se encuentra dentro de alguno de los agregados, es una relación entre éstos. De manera similar, el link de un item de la orden podría cruzarse con otra estructura de agregados para productos, de ser necesario. Pero como se quieren mostrar las órdenes y los clientes, se puede agregar los nombres de los productos dentro del agregado de órdenes. Esta denormalización es común para los agregados ya que varían según sea necesario para lograr minimizar el número de agregados accedidos durante la interacción.

Lo importante a destacar es que los límites de los agregados son puestos según sea necesario. Se debe analizar cómo se accederá a la información al momento de desarrollar la aplicación. Se podrían dibujar los límites de los agregados de maneras diferentes: por ejemplo, poniendo todas las órdenes para un cliente dentro del agregado de dicho cliente.

Como casi todo lo relativo a modelado, no hay una respuesta universal sobre como diseñar los límites de un agregado. Depende enteramente de cómo uno tiende a manipular la información. Si uno tiende a acceder a la información de un cliente junto con todas sus órdenes a la vez, entonces sería preferible un agregado único. Pero, si uno tiende a enfocarse en acceder a cada orden de manera particular, entonces sería preferible tener agregados separados para cada orden.

Naturalmente, depende completamente del contexto. Algunas aplicaciones preferirán una o la otra, incluso dentro del mismo sistema. Es por esto que mucha gente prefiere ignorar a los agregados.

### 3.4.2 Consecuencias de orientación a agregados y agreggate ignorant

Mientras que el mapeo relacional captura de manera adecuada los distintos elementos y sus relaciones, lo hace sin una noción concreta de entidades de agregados. Analizándolo se podría decir que una orden consistía de ítems, una dirección de envío y un pago, y en la base de datos se podría expresar en términos de claves foráneas y relaciones. Pero no hay nada que distinga las relaciones que representan agregados de aquellas que no lo hacen. Como resultado, la base de datos no puede usar el conocimiento de estructuras de agregados para ayudar a almacenar y distribuir la información. Existen varias técnicas de modelado de datos que aportan maneras de marcar estructuras de agregados o compuestas, pero raramente son usadas para proveer información sobre relaciones de agregados o compuestos.

Trabajando con bases de datos orientadas a agregados se tiene una semántica considerablemente más clara para enfocarse en la unidad de interacción con el almacenamiento de la información. Dependiendo de lo que el usuario considere que será conveniente según lo que necesitará la aplicación.

Las bases de datos relacionales no poseen, dentro de sus modelos de datos, el concepto de agregados. Por lo tanto **ignoran cualquier agregado que dentro de ellos se encuentre**. Lo mismo sucede con las bases de datos NoSQL basadas en grafos. Esto no es necesariamente malo ya que una estructura que ignore a los agregados permite buscar información en diversos sentidos más fácilmente. En ciertas situaciones es difícil delimitar los límites de un agregado si no se tiene una

clara idea de las funciones de la aplicación ya que éstos dependen principalmente de la forma en la que se va a acceder a la información. Tomando el ejemplo del sistema de ventas online, un agregado de órdenes puede ser muy provechoso cuando un cliente quiere ver que ha comprado o cuando se quieren procesar éstas, pero si se quisiera saber la cantidad de elementos vendidos de un producto habría que buscar entre cada agregado en la base de datos, lo que ocasionaría grandes retrasos. Por esto mismo, muchas veces es positivo que se ignoren los agregados.

El principal fundamento de ser de las bases de datos orientadas a agregados es que se adaptan perfectamente a correr en clusters, por lo cual cumplen con una de las principales ventajas de NoSQL en general. Si estamos corriendo la aplicación en clusters debemos minimizar la cantidad de nodos accedidos cuando buscamos información. Incluyendo explícitamente a los agregados, le estamos dando a la base de datos una importante señalización sobre que partes de los datos serán manipulados de manera conjunta y, por lo tanto, deberían agruparse en el mismo nodo.

Las bases de datos relacionales te permiten manipular cualquier combinación de filas desde cualquier tabla en una sola **transacción**. Dichas transacciones son las llamadas transacciones ACID (Atomic, Consistent, Isolated, Durable). De las características que poseen dichas transacciones, la atomicidad permite que muchas filas abarcando muchas tablas sean actualizadas como una sola operación. Esta operación o bien tiene éxito o falla en su totalidad, y las operaciones concurrentes son aisladas entre sí así no pueden acceder a actualizaciones parciales.

Los agregados también influyen de manera sustancial en las transacciones. Aunque se diga que las bases de datos NoSQL no soporten operaciones ACID sacrificando consistencia, en general lo que sucede es que no soportan dichas operaciones que abarquen muchos agregados, pero sí operaciones ACID que manejen de un agregado a la vez. Esto significa que si tenemos que manipular múltiples agregados de una manera atómica, debemos manejarlo desde el código de la aplicación. Igualmente, en la práctica se ve que la mayoría de las veces es posible mantener nuestra necesidad de atomicidad dentro de un solo agregado.

También cabe recordad que bases de datos NoSQL basadas en grafos y otras tantas que ignoren agregados no soportan transacciones ACID de la misma manera que lo hacen las bases de datos relacionales.

### 3.4.3 Agregados, clave-valor y basado en documentos

Las bases de datos clave-valor y las basadas en documentos son orientadas a agregados. Uno piensa en estas bases de datos como algo primordialmente construido a través de agregados. Ambos tipos de base de datos consisten en montones de agregados donde cada uno de ellos posee una clave o un ID que es usado para acceder a la información.

La diferencia entre ambos modelos es que en las bases de datos clave-valor, el agregado es opaco a la base, solo un gran conjunto de bits. Al contrario, una base de datos basada en documentos es capaz de ver la estructura de dicho agregado. La ventaja de la opacidad de las bases clave-valor es que podemos almacenar cualquier cosa que queramos en el agregado. La base de datos podría

imponer algún límite de tamaño general pero en líneas generales hay una completa libertad. Una base de datos basada en documentos impone límites en lo que podemos almacenar, definiendo estructuras y tipos permitidos. A cambio de esto, se obtiene una mayor flexibilidad en el acceso.

Con una base de datos clave-valor solo podemos acceder a un agregado por una búsqueda basada en su clave. Con una base de datos basada en documentos, podemos buscar en ello basándonos en los campos del agregado, podemos obtener parte del agregado en vez del agregado completo, y la base de datos puede crear índices basados en los contenidos del agregado.

En la práctica, la diferencia entre ambos tipos de bases de datos es un poco borrosa. Los desarrolladores a veces ponen un campo ID en una base de datos basada en documentos para hacer una búsqueda al estilo clave-valor. Bases de datos clasificadas como clave-valor pueden permitirte estructuras de datos más allá de un agregado opaco. Ejemplo Riak, REdis.

A pesar de lo borroso que pueden resultar los límites, la distinción general se mantiene. Con bases de datos clave-valor, se espera que la mayoría de las búsquedas se realicen por su clave. Con bases de datos basadas en documentos, se espera que se realicen búsquedas basadas en la estructura interna del documento, el cual podría ser una clave, pero es más probable que sea cualquier otra cosa.

### 3.4.4 Resumen Agregados

Lo que comparten es la noción de un agregado indexado por una clave que se puede usar para búsquedas. Este agregado es fundamental para correr en un cluster ya que la base de datos se asegurará que toda la información de un agregado se encuentre alojada en un mismo nodo. El agregado también sirve como la unidad atómica para las actualizaciones, aportando un control transaccional útil pero limitado.

Dentro de esa noción de los agregados, existen algunas diferencias.

El modelo clave-valor trata a los agregados como un conjunto opaco, o sea que solo se puede hacer una búsqueda por clave para todo el agregado. No se puede hacer una búsqueda por o devolver una parte del agregado.

El modelo basado en documentos hace al agregado transparente a la base de datos permitiendo hacer búsquedas y devoluciones parciales. Sin embargo, ya que el documento no tiene ningún esquema asociado, la base de datos no puede trabajar mucho en la estructura del documento para optimizar el almacenamiento y la devolución de las partes del agregado.

Los modelos basados en familias de columnas dividen el agregado entre conjuntos de estas columnas, permitiendo a la base de datos tratarlos como unidades de información dentro del agregado de la fila. Esto impone cierta estructura en el agregado pero permite a la base de datos tomar ventaja de esta estructura para mejorar la accesibilidad.

# Capítulo 4

# Los desafíos de desarrollar aplicaciones móviles y modelos de distribución

## 4.1 Los desafíos de desarrollar aplicaciones móviles

El exponencial aumento del uso de aplicaciones móviles que se ha visto en los últimos años impuso un nuevo paradigma a la hora de desarrollar sistemas distribuidos. Los dispositivos móviles no sólo son computadoras más pequeñas sino que trajeron consigo una gama de restricciones nuevas a tener en cuenta a la hora de desarrollar aplicaciones [9].

La computación móvil está caracterizada principalmente por cuatro restricciones:

1. *Los elementos móviles son relativamente pobres en recursos*:
   Para un costo y nivel de tecnología dados, las consideraciones de peso, potencia, tamaño y ergonomía exigen consecuencias en recursos computacionales como velocidad del procesador, tamaño de la memoria y capacidad del disco. Si bien la brecha ha disminuido considerablemente, los elementos móviles siempre serán relativamente pobres en recursos, comparados con elementos estáticos.

2. *La movilidad es inherentemente arriesgada:*
   Además de las preocupaciones básicas sobre seguridad, los dispositivos portátiles son más vulnerables a pérdidas o daños. Es más probable que un ejecutivo sea asaltado en la calle y le roben su dispositivo a que entren a la oficina y le roben la computadora.

3. *La conectividad móvil es altamente variable en performance y confianza:*
   Algunos lugares pueden ofrecer conectividad inalámbrica confiable de alta velocidad mientras otros pueden sólo ofrecer una conectividad lenta. En la calle incluso, un cliente móvil debe confiar en redes inalámbricas de baja velocidad con huecos en la cobertura. También hay que tener en cuenta que el servicio de redes móviles no es parejo en todos los países, cambiando drásticamente la calidad de servicio entre unos y otros.

4. *Los elementos móviles se basan en una finita fuente de energía:*
   A pesar de que las baterías para dispositivos inalámbricos han mejorado sustancialmente, la necesidad de ser sensible al consumo de energía no disminuirá. La preocupación por el consumo de energía debe abarcar muchas capas de hardware y software para ser realmente efectiva.

La movilidad del dispositivo intensifica la tensión entre autonomía e interdependencia, que es característica de todos los sistemas distribuidos.

La relativa pobreza de recursos de los dispositivos móviles así también como su relativa baja confianza dificulta la dependencia en servidores estáticos. Pero la necesidad de lidiar con redes poco confiables y de baja velocidad, así también como la necesidad de ser sensibles al consumo de energía dificulta también dependencia en el mismo dispositivo.

Cualquier enfoque viable a la computación móvil debe manejar un balance entre estas preocupaciones. Este balance n puede ser estático: mientras vayan cambiando las circunstancias de un cliente móvil, se debe reaccionar y reasignar dinámicamente las responsabilidades del cliente y el servidor. En otras palabras, debe ser adaptativo.

## 4.2 Mobile Cloud Computing

Como se ha visto, si bien el rápido progreso de la computación móvil la ha convertido en una fuerte tendencia en el desarrollo de tecnología, ésta aún enfrenta muchos retos en cuanto a recursos y comunicaciones. Los limitados recursos impiden significativamente la mejora de la calidad de los servicios [10].

Para superar esta falta de surgieron soluciones como el Cloud Computing, la cual ha sido aceptada por muchos como la infraestructura de la nueva generación. Ésta ofrece algunas ventajas al permitir a los usuarios usar infraestructura (servidores, redes, almacenamiento), plataformas (middleware, sistemas operativos) y software (aplicaciones) provistas por proveedores en la nube a un bajo costo, como por ejemplo Google y Amazon.

Además, Cloud Computing permite a los usuarios utilizar recursos elásticamente bajo demanda. Como resultado, las aplicaciones móviles pueden ser rápidamente provistas y distribuidas con un mínimo esfuerzo de mantenimiento y una mínima interacción con el proveedor.

Con la explosión de las aplicaciones móviles y el soporte por parte de Cloud Computing para con una variedad de servicios para usuarios móviles, el Mobile Cloud Computing (MCC) es introducido como una integración del Cloud Computing en los ambientes móviles. El MCC trae nuevos tipos de servicios y facilidades para que usuarios móviles aprovechen todas las ventajas de Cloud Computing.

MCC es una infraestructura donde tanto el almacenamiento de información como el procesamiento de los datos ocurren fuera del dispositivo móvil. Las aplicaciones móviles en la nube mueven el poder de procesamiento y el almacenamiento de la información fuera de los dispositivos hacia plataformas de procesamiento poderosas y centralizadas localizadas en la nube. Estas aplicaciones centralizadas son luego accedidas a través de una conexión inalámbrica basada en un cliente nativo o un navegador web en el dispositivo móvil.

En pocas palabras, MCC provee a los usuarios de tecnologías móviles el procesamiento de información y los servicios de almacenamiento en la nube. Los dispositivos móviles no necesitan un gran conjunto de recursos de hardware ya que todos los módulos de cómputo pesado pueden ser procesados en la nube.

MCC tiene muchas ventajas y es una de las soluciones más aceptadas y usadas en la actualidad. Sin embargo, al integrar diferentes campos como el cloud computing y las redes móviles, debe enfrentarse a muchos desafíos técnicos.

Entre aquellos desafíos se encuentran:

1. *Bajo ancho de banda:*
   La velocidad del ancho de banda es uno de los grandes problemas en MCC ya que el radio de recursos para redes móviles es más escueto en comparación con las redes tradicionales.

2. *Disponibilidad:*
   La disponibilidad del servicio se convierte en un tema mucho más importante en MCC que en cloud computing con redes tradicionales. Los usuarios móviles pueden no ser capaces de conectarse a la nube para obtener el servicio debido a congestión de tráfico, fallas en la red o estar fuera del alcance de la señal.

3. *Heterogeneidad:*
   MCC se usará en una amplia gama de redes móviles heterogéneas. Distintos nodos móviles accederán a la nube a través de diferentes tecnologías de acceso. Como resultado, surge el problema de cómo manejar la conectividad inalámbrica mientras se satisfacen los requerimientos de MCC.

4. *Competencia por los recursos:*
   Con una cantidad creciente de servicios en la nube, la demanda para acceder a esos recursos también crece. Como resultado, un método para trabajar con los recursos en la nube y tratar la competencia por ellos se vuelve algo significativo. Éste no es un problema simple si a estos retos se le agregan los mencionados anteriormente.

## 4.3 Funcionalidad offline

La funcionalidad offline es un factor importante que necesita ser tenido en cuenta cuando se diseña una aplicación móvil. En los últimos años, las conexiones de datos y las redes móviles se han extendido ampliamente en varios países y las aplicaciones web requieren típicamente una conexión activa a internet para funcionar [8, CAPÍTULO 2].

Sin embargo, las conexiones de datos o WiFi son fácil y comúnmente interrumpidas en ciertos lugares (sótanos, subterráneos) o simplemente no disponibles (aviones, lugares alejados). Incluso cuando están disponibles, hay otros factores a tener en cuenta como cargos de roaming, poco ancho de banda o consumo de batería. El usuario incluso puede deshabilitar la conexión en cualquier momento.

Por lo tanto, diseñar una aplicación que se pueda usar y funcione correctamente offline puede ser crucial.

Un buen ejemplo es un caso simple donde una persona quiera mantener un registro de sus comidas en una aplicación que sincronice los datos con un servidor remoto. Idealmente se debería realizar la tarea aunque no haya conexión a internet en el momento, para que luego la data sea sincronizada cuando esté disponible una conexión confiable.

Una aplicación web típica es almacenada en un servidor remoto, por lo que tradicionalmente sus páginas no pueden ser accedidas sin una conexión a internet. El cache del navegador puede ser usado para almacenar una pequeña cantidad de datos, pero esto no suficiente para la mayoría de los casos.

El nuevo estándar HTML5 especifica dos mecanismos que solucionan este problema: una API de la base de datos basada en SQL para almacenar datos localmente; y un cache de aplicación HTTP

offline para asegurarse que las aplicaciones estén disponibles incluso cuando el usuario no esté conectado a la red.

Si bien aún es soportado por la mayoría de los navegadores, el W3C ha dejado de trabajar en la base de datos web SQL por falta de implementaciones independientes por lo que se lo considera *deprecated*.

Como ya se ha descripto anteriormente, las aplicaciones nativas pueden ser descargadas y guardadas en el dispositivo, simplificando la funcionalidad offline. Además, tienen acceso a los componentes del mismo como la cámara, el GPS o el acelerómetro, y pueden también guardar grandes cantidades de información en el *filesystem*.

Las aplicaciones híbridas pueden sacar ventaja de esto gracias a su capa nativa, lo que les permite interactuar con el dispositivo justo como lo haría una aplicación nativa, incluso guardando datos en la base de datos local.

Las aplicaciones pueden sacar ventaja del almacenamiento local de diferentes maneras. Algunos permiten a sus usuarios marcar el contenido que desean tener disponible offline, mientras otros lo pueden hacer automáticamente en background, para una experiencia más fluida.

Trabajar con aplicaciones offline puede ser una ventaja la mayoría del tiempo, pero tiene sus desventajas. Por ejemplo, ya que los datos solo son sincronizados cuando haya disponible una conexión de red, no se puede garantizar que la información este completamente actualizada en un cierto momento. Muchos clientes pueden actualizar la misma información, por lo que el usuario puede no siempre tener la última versión. Gracias a esto, se debe tener en cuenta los casos donde los clientes actualicen la misma información simultáneamente, una situación que puede llevar a conflictos e información desconfiable.

## 4.4 Cuestiones comunes a las capas de almacenamiento de apps móviles

Comparándolas con aplicaciones web clásicas y de varios niveles, las aplicaciones móviles tienen requerimientos comunes y específicos en lo concerniente a la persistencia y el procesamiento de datos. En dichas aplicaciones, las características de las bases de datos deben ser analizadas distintivamente para las capas cliente y servidor. Siendo minimalistas, aisladas y sólo en memoria en la primera; y centralizadas, distribuidas, sincronizadas, basadas en discos y con gran recolección de datos en la segunda [11].

Las aplicaciones móviles comparten a su vez muchas características de la arquitectura de aplicaciones cliente/servidor o aplicaciones web de varios niveles **(Figura 6)**. Los niveles de persistencia tienen requerimientos similares en términos de almacenamiento, lenguaje de definición de datos (Data Definition Language), lenguaje de manipulación de datos (Data

Manipulation Language) y queries. Además, los mecanismos de accesibilidad de las capas de aplicación son similares a otras aplicaciones web.

La mayor diferencia de restricciones entre bases de datos de aplicaciones móviles y webs es que mientras que las capas de persistencia de los servidores son similares, las plataformas clientes de aplicaciones móviles tienen una menor escala de recursos disponibles. Muchas aplicaciones móviles requieren mecanismos de replicación y sincronización para la persistencia de datos de un smartphone con una base de datos de gran escala centralizada disponible en un servidor.

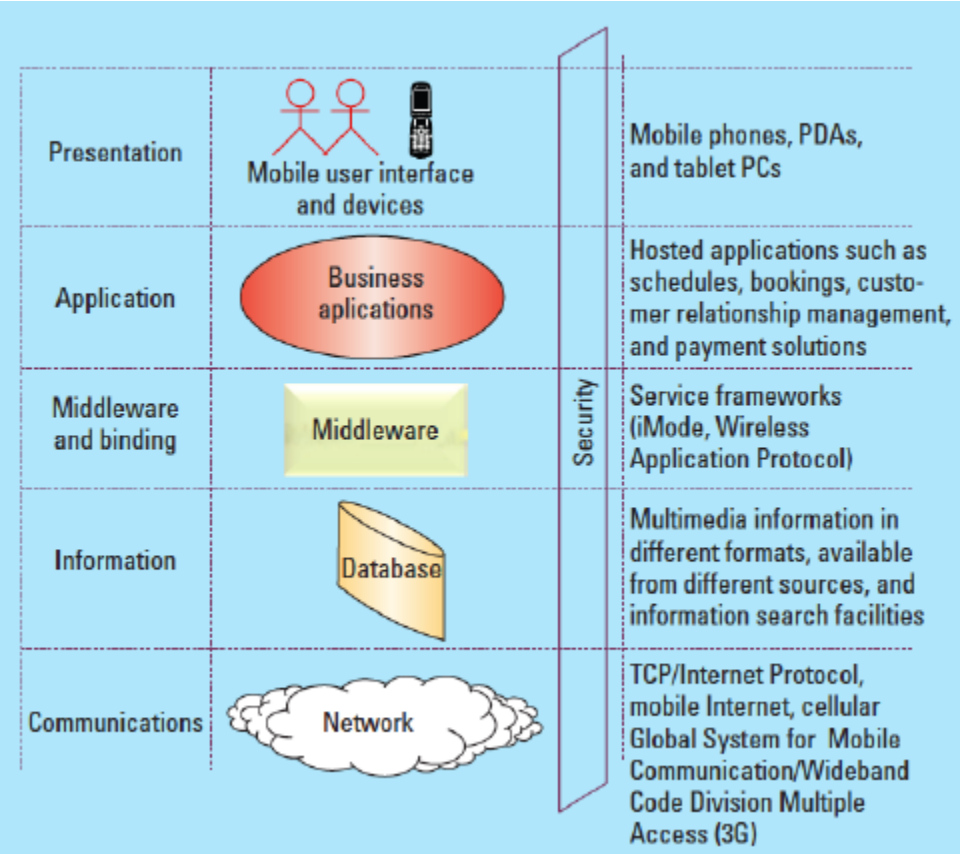


*Figura 6 Arquitectura de aplicaciones móviles*

Los mecanismos de bases de datos móviles, como queries, indexaciones, persistencia, deben ser separados de los DBMS monolíticos tradicionales y estar disponibles como componentes integrables (por ejemplo los DLLs) dentro de las aplicaciones [12].

Dichos DBMS móviles e integrables deben cumplir, completa o parcialmente, con las siguientes características:

1. *Integrables en aplicaciones:*
   Los DBMS móviles conforman una parte integrable de la aplicación o la infraestructura de la aplicación, a veces incluso sin requerir administración. La funcionalidad de la base de datos es manejada como parte de la aplicación. Mientras que la base de datos debe ser integrable como un DLL en la aplicación, también debe ser posible desplegarla como un DBMS que soporte múltiples transacciones y aplicaciones en sí mismo.

2. *Pequeño tamaño:*
   Para muchas aplicaciones, especialmente aquellas que son descargables, es importante minimizar el tamaño del DBMS. Ya que la base de datos es parte de la aplicación, el tamaño del DBMS afecta al de la aplicación en sí. Además, es deseable que el código de ejecución sea sencillo para lograr una mejor eficacia de ejecución. Muchas de estas aplicaciones no requieren la funcionalidad completa de un DBMS comercial; solamente requieren queries y ejecuciones simples en ambientes limitados.

3. *Ejecutable en ambientes móviles:*
   El DBMS que corre que ambientes móviles tiene a ser una versión especializada de otros DBMS. Además de manejar las limitaciones de memoria, disco y procesador de estos dispositivos, el DBMS debe correr también en sistemas operativos especializados. El DBMS debe ser capaz de almacenar y reenviar información a las bases de datos back-end ya que la sincronización con estos sistemas es crítica.

4. *DBMS dividido en componentes:*
   Muchas veces, para soportar el requerimiento de tamaño, es importante incluir solo la funcionalidad requerido por las aplicaciones. Por ejemplo, muchas aplicaciones simples sólo requieren métodos de acceso secuencial (ISAM) mientras que otras solo requieren acceso XML y no relacional. Por lo que no es necesario incluir el procesador de queries aumentando el tamaño de la aplicación. Por esto mismo, debería ser posible elegir los componentes deseados.

5. *DBMS auto-manejable:*
   El DBMS integrado es invisible al usuario de la aplicación. No puede haber un administrador que maneje la base de datos ni que pueda iniciar operaciones como backups y recoverys. La base de datos debe ser auto manejada o manejada por la aplicación. Además, los DBMS integrados deben auto instalarse con la aplicación y no ser instalados explícita o independientemente. De manera similar, cuando la aplicación se encuentre apagada, el DBMS debe apagarse también.

6. *DBMS en memoria:*
   Algunas aplicaciones requieren una alta disponibilidad en datastores lo suficientemente pequeños para ser contenidos en memoria. DBMS contenidos enteramente en memoria

requieren procesadores de query especializados y técnicas de indexación optimizadas para uso en memoria principal. Dichos DBMS también pueden soportar datos que puedan nunca ser persistidos.

7. *Base de datos portable:*
   Existen muchas aplicaciones que requieren un despliegue muy simple e instalar la aplicación debería instalar la base de datos asociada. Esto hace que la base de datos sea altamente portable. No debería haber necesidad de instalar el DBMS separadamente. Instalar la aplicación debería instalar el DBMS y copiar el archivo de la base de datos completaría la migración de datos.

8. *Base de datos sin código:*
   Las bases de datos portables deben ser también seguras. Código ejecutable puede acarrear virus y otras amenazas. Eliminando cualquier código almacenado en la base de datos, hace que ésta sea más segura y portable.

9. *Sincronizable con data sources del back-end:*
   En el caso de escenarios móviles, debe ser posible sincronizar la información con los data sources del back-end. En casos cliente/servidor, la información es captada de la base de datos del back-end al cache, procesada, y luego sincronizada con la base de datos back-end.

10. *Manejo remoto:*
    A pesar de que los DBMS en ambientes móviles tengan que ser auto-manejables, es importante permitir además que sean manejados remotamente. En diversas empresas, los dispositivos móviles deben ser configurados y manejados de manera que siga los estándares de la compañía. Por lo tanto, un manejo remoto y centralizado de estos dispositivos es necesario.

11. *Interfaces de programación personalizables:*
    Un importante uso de DBMS integrables es en aplicaciones centradas en datos especializados. Dichas aplicaciones usan una variedad de data models y lenguajes de consulta. Dichos DBMS deben ser manejado en componentes y extensible para permitir lenguajes de programación específicas al dominio.

# Capítulo 5

# Replicación

## 5.1 Replicación

Como ya se ha comentado, NoSQL aporta muchas ventajas en el manejo de grandes bases de datos en cuanto a performance. En los últimos años, con la aparición y la expansión masiva de diversos dispositivos capaces de acceder a internet, las aplicaciones con gran flujo de datos como las redes sociales y los motores de búsqueda han forzado al extremo a los RDBMS tradicionales, al menos ejecutándose en un único nodo.

Los servicios de compañías web (Google, Amazon, etc) llegaron a tener mucha información como para que pueda ser manejada por una simple computadora por lo que éstas desarrollaron arquitecturas basadas en clusters. De aquí han surgido modelos de Cloud Computing y Mobile Cloud Computing, de los que ya hemos hablado, con todas sus ventajas y desventajas.

Sin embargo, cuando hablamos del manejo de aplicaciones en móviles y su funcionamiento con Mobile Cloud Computing hay un problema que surge inherentemente, la conectividad. Una de las primeras restricciones que afrontan este tipo de dispositivos es que la conectividad móvil es altamente variable en performance y confianza. Los dispositivos móviles no pueden garantizar una conexión constante e incluso contando con redes móviles existen problemas de conectividad, habiendo momentos donde la cobertura pueda ser baja o inexistente, especialmente en países en desarrollo.

Del mismo modo, a pesar de todas las ventajas que presentan las bases de datos NoSQL en cuanto a velocidad y escalabilidad, y por lo que se pudo comprobar en "Benchmarking Embedded Mobile

Databases", tener presente DBMS capaces de manejar BigData de manera nativa y offline en un dispositivo móvil no presenta de por sí una ventaja apreciable, sino todo lo contrario.

También se deben tener en cuenta las inherentes limitaciones de recursos de los dispositivos móviles que dificultan el manejo de datastores NoSQL en estos dispositivos. Si bien existen adaptaciones para estos ambientes, como CouchBase Lite por ejemplo, éstos por lo general son versiones ligeras y no presentan el espectro completo de instrucciones disponibles en otras versiones.

Por todas estas razones, no es del todo conveniente manejar NoSQL en ambientes móviles con un enfoque completamente nativo u otro completamente orientado a Cloud Computing. Surge entonces como alternativa el concepto de replicación.

### 5.1.1 ¿Qué es la replicación?

Replicación consiste en una copia frecuente de datos desde una base de datos en una computadora o servidor a otra base de datos para que todos los usuarios compartan el mismo nivel de información. El resultado es una base de datos distribuida en la cual los usuarios puedan acceder a información relevante a su tarea sin interferir con el trabajo de otros [13].

Un sistema de manejo de bases de datos distribuidas (DDBMS) se asegura que todos los cambios y borrados realizados en la información guardada en cualquiera de las locaciones sean automáticamente reflejadas en la información almacenada en las otras locaciones. Por lo tanto, cada usuario siempre ve en su copia de la base de datos información que es consistente con la información que ven otros usuarios.

Muchas veces, y especialmente en dispositivos móviles, cuando se tienen problemas con manejo de Bigdata el cuello de botella en el flujo de información no es la información en sí sino la carga de la misma. Aunque la cantidad de información requerida no supere la capacidad de procesador, quizás el número de requerimientos si lo haga. Una de las soluciones de ese problema es la replicación.

Como ya se ha comentado en las características relativas a la capa de almacenamiento en aplicaciones móviles, un DBMS móvil debería ser capaz de sincronizarse con data sources del back-end. En el caso de escenarios móviles, debe ser posible sincronizar la información con data sources en la nube. De esta forma, se puede tener en dispositivos móviles bases de datos NoSQL (o porciones de ella) que se almacene y procese como bases de datos locales y nativas pero que se replique a un servidor.

### 5.1.2 ¿Por qué usar replicación?

Como ya se ha dicho, la conectividad móvil es altamente variable en performance y confiabilidad, lo que conlleva a situaciones donde se deben realizar operaciones sin conexión. Por lo tanto, los

usuarios de aplicaciones móviles, requieren copias locales de la información importante de la base de datos ya que están comúnmente desconectados de la red [14, CAPÍTULO 1].

Esta situación se podría ejemplificar con el siguiente escenario. Imaginen un vendedor que viaje de cliente a cliente recolectando órdenes. El vendedor colecta toda la información en una tablet que también usa para acceder a información que indique cuánto tiempo tomarán esas órdenes y otra información como condiciones posibles de la orden. Cuando el vendedor se encuentra visitando a un cliente, la mayoría del tiempo la comunicación entre la tablet y la base de datos corporativa central, donde toda la información requerida debe ser ingresada y accedida, no es posible. Incluso aunque la tablet posea redes móviles como 3G o 4G, la comunicación se puede cortar por deficiencia de la red o puede hallarse en un lugar alejado del área de cobertura. En este momento tiene sentido la replicación de datos. Como se ha marcado, la información necesaria tiene que ser copiada en el dispositivo del vendedor para poder proveer la funcionalidad requerida. Periódicamente, el vendedor debe enviar las nuevas órdenes al sistema central por alguno de los medios de comunicación. Cuando se reconecte a la base de datos corporativa o algún otra aplicación es cuando los dos datasets deben ser sincronizados.

Otro buen ejemplo de uso sería en situaciones donde se debe mantener información sincronizada entre dispositivos, pero no es o no debería ser posible una conexión constante. Por ejemplo, el servicio en la nube iCloud presentado por Apple en 2011. Este servicio se usa para intercambiar información entre los dispositivos móviles y desktop de Apple, incluyendo calendarios, contactos, fotos y backup de los dispositivos móviles. iCloud también incluye APIs de almacenamiento para desarrolladores de iOS y OS X con el propósito de sincronizar estados de aplicaciones y demás contenido de usuario entre dispositivos. Un usuario que tenga un iPhone y un iPad y que use la misma aplicación en ambos dispositivos debería poder cambiar entre uno u otro sin notar el cambio. iCloud funciona como una base de datos de almacenamiento local embebido que se replica con iCloud. De todas formas, se restringe a aplicaciones nativas que corran en el sistema operativo móvil o desktop de Apple [15].

La replicación se usa para lograr una mejor disponibilidad y performance usando múltiples copias de sistema servidor, donde la disponibilidad sea el mayor objetivo. Si una de las copias o réplicas no funciona, el servicio seguirá funcionando ya que es provisto por un conjunto de éstas. También es muy útil si el link de comunicación entre dos sistemas sólo está disponible intermitentemente, cosa que sucede muy a menudo en aplicaciones móviles [14, CAPÍTULO 3].

Hasta ahora, no hay muchas aplicaciones que usen replicación, pero la mayoría de ellas funciona pobremente o directamente no funciona cuando no hay conexión a internet. La replicación de base de datos permite una nueva clase de aplicaciones móviles, especialmente en áreas donde la habilidad de utilizar la aplicación independientemente de la ubicación y la conexión a internet son importantes [15].

### 5.1.3 Tipos y técnicas de replicación

Hay principalmente dos tipos de replicación, sincrónica y asincrónica:

La replicación sincrónica escribe la información en los sitios primarios y secundarios al mismo tiempo para que ésta esté disponible en ambos. Es más costosa que otras formas de replicación ya que introduce latencia que hace más lenta la aplicación principal. Es por lo general usada con propósitos de recuperación de desastres y es preferible en aplicaciones con uso no muy frecuente que recovery pero que no pueda permitirse perder información [16].

En cambio, la replicación asincrónica escribe la información en el arreglo de almacenamiento principal primero y, dependiendo de la implementación, hace commit a la información a ser replicada en memoria o disco. Luego copia la información en tiempo real o a intervalos programados. No requiere tanto ancho de banda como la replicación sincrónica y puede tolerar algo de degradación en la conectividad [17].

Existen muchas implicaciones técnicas relacionadas a la replicación de datos. El principal requisito en la replicación de datos es que las réplicas deberían comportarse funcionalmente como servidores que no estén replicados. Este requerimiento no puede cumplirse siempre ya que uno de los mecanismos para solucionar este problema, la replicación sincrónica, produce una alta carga en transacciones. Por lo tanto, la replicación sincrónica es considerada muy difícil de lograr de una manera eficiente. Por otro lado, utilizando replicación asincrónica y actualizando las réplicas independientemente, se puede lograr en cierto punto un estado común e idéntico de manera más eficiente. Si bien esta solución puede llevar a situaciones donde diversas transacciones actualicen la misma información generando conflictos, se puede solucionar marcando el tiempo de las órdenes de replicación [14, CAPÍTULO 3].

Mantener el orden de las actualizaciones requiere sincronización, lo que impone implicaciones de performance que requieren técnicas de sincronización sofisticadas, resultando en un alto grado de complejidad. La replicación, por lo tanto, impone un constante intercambio entre consistencia y eficiencia.

Las diversas técnicas de replicación existentes pueden ser caracterizadas identificando dos dimensiones diferentes. Primero, dónde se puede llevar a cabo una actualización, en un servidor dedicado o en cualquiera dentro del cluster de réplicas; y segundo, si una actualización será hecha sincrónica o asincrónicamente en todas las réplicas. Hay dos principales técnicas, la replicación con un sólo nodo maestro (*Single Master Replication*) y la replicación con múltiples nodos maestros (*Multi Master Replication*).

Cuando se utiliza replicación con un sólo nodo maestro, una réplica es designada servidor donde se aplican las transacciones de actualización. Esta réplica se llama réplica primaria y las actualizaciones en ésta son luego distribuidas a las otras réplicas, llamadas réplicas secundarias. Durante el proceso de distribución, el orden en cuál las transacciones son ejecutadas en el nodo primario se preserva y todas las réplicas deben atravesar la misma secuencia de estados, por lo

tanto se puede ejecutar cualquier query sobre cualquier réplica entre dos actualizaciones. Ya que hay sólo un servidor que pueda ser actualizado, no pueden surgir conflictos debidos a actualizaciones simultáneas en diferentes réplicas. El único requerimiento que tiene que ser cumplido en cualquier momento es que sólo haya un nodo primario. Los secundarios, sin embargo, también pueden ser backups del primario e incluso es posible tener primarios para subconjuntos específicos de la información que deban ser replicados.

Mientras tanto el principal como los servidores secundarios estén disponibles, esta técnica funcionará sin problemas. En caso de falla, se debe aplicar un comportamiento especial dependiendo de la ubicación de dicha falla. Si falla un nodo secundario, el sistema continúa su trabajo normalmente; y cuando éste se recupera, debe detectar qué actualizaciones han tenido lugar desde que falló. Si falla el servidor principal, lo primero que necesita el sistema es elegir un nuevo nodo principal. Una réplica inicializa el algoritmo de elección tomando los ids de las demás y elige aquella que tenga el número más alto para luego indicarles a las demás cuál es el nuevo maestro. Una vez elegido un nuevo maestro, todas las réplicas deben encontrarse en el mismo estado y para evitar una situación el nuevo maestro puede detener el procesamiento de transacciones hasta que todos los nodos hayan reconocido su actualización; por lo que encontrar un nuevo nodo primario y reconfigurar todas las réplicas puede ser bastante costoso y si sucede muy seguido puede degradar significativamente la performance del sistema.

La replicación con múltiples nodos maestros puede entenderse como un sistema de réplicas con múltiples servidores primarios definidos como con la técnica replicación con un sólo nodo maestro. Los múltiples servidores primarios manejan las actualizaciones independientemente entre sí mientras están desconectados y cuando se reconectan a la red se intercambian las actualizaciones grabadas. Es algo común que una red se divida en particiones donde cada una de éstas siga procesando actualizaciones, un ejemplo claro sería cuando se maneja la base de datos desde dispositivos móviles no siempre conectados junto con un servidor central. Por lo tanto una técnica de manejo de replicación con un sólo nodo servidor no funcionaría en este caso.

Durante el período de tiempo en el cual los dos sistemas están desconectados pueden actualizarse equivalentes ítems en ambos, lo que resultaría en conflictos cuando dichos sistemas intenten unir sus datos. Igualmente, los conflictos pueden evitarse en el diseño de las aplicaciones, usando por ejemplo timestamps.

### 5.1.4 Replicación en NoSQL

En los últimos años, las bases de datos NoSQL se han hecho extremadamente populares con web services. Su principal ventaja sobre las bases de datos relacionales incluyen la escalabilidad de la carga de datos y mejor performance sobre grandes data sets. Las bases de datos NoSQL están diseñadas para arquitecturas distribuidas y permiten una replicación entre nodos más simple que la de las bases de datos relacionales [15, CAPÍTULO 1].

A pesar de sacrificar características de consistencia bien establecidas, la mayoría de las bases de datos NoSQL proveen lo que se dice consistencia eventual (*eventual consistency*), donde la base de datos puede estar momentáneamente en un estado inconsistente pero lo estará eventualmente, debido en parte a la propagación de actualizaciones a través de los clusters.

Las bases de datos clave-valor, las más simples en estructura, generalmente proveen mecanismos de persistencia y funcionalidad adicional como replicación junto con versionado, bloqueos, etc [19].

*Riak* soporta replicación de objetos y sharding por hash en la clave principal; permite a los valores de réplica ser temporalmente inconsistentes. La consistencia se puede configurar especificando cuantas réplicas en diferentes nodos deben responder por una lectura exitosa y cuántas por una escritura exitosa. *Tokyo Cabinet* maneja replicación asincrónica con maestro dual o maestro-esclavo. La recuperación de un nodo fallido es manual y no hay sharding automatizado. *Redis* provee replicación sincrónica y actualizaciones atómicas por bloqueo. *Scalaris* maneja replicación sincrónica y la información está garantizada de ser consistente ya que la mayoría de las copias deben ser actualizadas antes de que se complete la operación. La información se almacena en memoria, pero la replicación y la recuperación de fallos de nodos proveen durabilidad de las actualizaciones. Scalaris usa un anillo de nodos, una distribución y estrategia de replicación inusual, y las lecturas y escrituras deben propagarse a la mayoría de las réplicas antes que una operación se complete.

Mientras Scalaris usa replicación sincrónica, el resto usa replicación asincrónica. Además, Voldemort, Riak, Tokyo Cabinet y sistemas Memcached pueden almacenar información en RAM o en disco, mientras los otros almacenan la información en la memoria RAM y proveen discos como backup o confían en replicación y recuperación por lo que no es necesario un backup.

Las bases de datos basadas en documentos, más complejas que las bases de datos clave-valor, también proveen mecanismos de replicación entre otras cosas.

*SimpleDB* soporta replicación asincrónica con consistencia eventual. *MongoDB* maneja replicación maestro-esclavo con recuperación automática. Esta replicación es asincrónica para una mayor performance, por lo que algunas actualizaciones pueden perderse en una caída del sistema.

*CouchDB* es especialmente popular por su capacidad de replicación maestro-a-maestro. Tiene un mecanismo de replicación de datos tolerante a fallas integrado. Maneja tanto replicación maestro-a-maestro y maestro-a-esclavo que puede ser usado entre instancias de CouchDB diferentes y geográficamente separadas o entre instancias simples [18, CAPÍTULO 1].

CouchDB logra la escalabilidad gracias a la replicación asincrónica y no a través de sharding. Las lecturas pueden ir a cualquier servidor, si no interesa tener siempre los últimos valores, y las actualizaciones deben ser propagadas a todos los servidores.

Los documentos en las bases de datos CouchDB se almacenan en una dirección de memoria y cada uno de éstos es identificado por un ID único. No hay correlaciones entre cada uno de estos documentos independientes. Debido a ésto, CouchDB introduce un nuevo modelo de filtrado y búsqueda de datos, donde la información es recuperada de la base de datos con el uso de vistas y/o agregados. Los agregados y las vistas actuarían como la parte de mapeo de un sistema MapReduce.

En el caso de las bases de datos tabulares, *Cassandra* también presenta particionamiento y replicación. Similar a otras bases de datos tabulares en modelo de datos y funcionalidad básica, tiene grupos de columnas, las actualizaciones se guardan en cache en memoria para luego cargarse al disco [19].

## 5.2 Aplicaciones reales de replicación junto con NoSQL en ambientes móviles

En este capítulo se abarcó de forma teórica la replicación en las bases de datos, poniendo énfasis en sus usos, en las distintas técnicas existentes y en cómo se utilizaría este recurso en bases de datos NoSQL. Sin embargo, esta es una técnica que ya se ha usado y sería más fácil explicar su utilidad con ejemplos prácticos.

Para ello se estudiaron tres casos con diversos resultados que ejemplificarían algunos de los muchos usos que puede tener la utilización de replicación en bases de datos NoSQL. El primero muestra un sistema de localización y reconocimiento de automóviles utilizando una red de sensores distribuidos en la vía pública; el segundo la actualización de una aplicación utilizada por inspectores que revisan edificios en construcción con conexión a internet limitada; y por último, el tercero el desarrollo de una app que puedan bajarse y utilizar en su vida diaria los usuarios de un sistema de monitoreo de salud.

### 5.2.1 Sistema de localización de automóviles utilizando sensores

Muchos de los logros de la ciencia y la tecnología han sido utilizados exitosamente por los servicios de las fuerzas de orden público para mejorar el índice de detección de crímenes y mejorar la seguridad pública. Uno de aquellos crímenes que podrían ser detectados y evitados usando avances científicos recientes es el robo de autos. Métodos automáticos de reconocimiento de patentes de autos en movimientos basados en un monitoreo distribuido de áreas urbanas podrían ayudar a encontrar dichos autos y reducir el número de robos [18].

El problema requiere alguno de los muchos dispositivos móviles capaces de adquirir la información visual. Si bien se podrían utilizar alguno de los ya establecidos sistemas de manejo de sensores, como TinyDB o SwissQM, éstos se basan en usar dispositivos demasiado sencillos para colectar la información cruda del ambiente, dejando el procesamiento al servidor. Si se tienen en cuenta los rangos de comunicación limitados, el manejo de consistencia en la red, la optimización de

procesamiento, el uso de energía, etc; el sistema no sería eficiente a gran escala. Un sistema de monitoreo distribuido que pueda procesar exitosamente complejos análisis de información, como reconocer patentes de autos moviéndose por un área urbana grandes, requiere otro enfoque. La cantidad de información visual colectada por un gran número de dispositivos no puede ser transferida a un sistema de procesamiento central y no puede ser analizada de una manera centralizada.

Un enfoque diferente asumiría un procesamiento de datos distribuido. Los dispositivos móviles ya no se ven limitados solamente a funcionar como unidades recolectoras de información, sino que son capaces de ejecutar análisis de datos complejos y notificar al servidor central del sistema sólo cuando se detecta información importante. El servidor central sería responsable solamente de definir tareas para los dispositivos móviles y de recolectar la información final. Incluso, no sería necesario que éste funcione para que trabajen los dispositivos móviles, podría ser apagado temporalmente sin tener impacto en el sistema. La creación de un sistema así es posible debido al aumento significativo del poder computacional de dispositivos móviles accesibles observado en los últimos años.

La tarea de localizar los autos requiere un algoritmo para identificar los números de las patentes de los autos sobre imágenes recolectadas por los dispositivos móviles, los cuales ya han sido estudiados y desarrollados extensivamente en los últimos años. Sirve en esta situación para mostrar como ejemplo que una tarea de procesamiento compleja puede ser ejecutada exitosamente en sensores móviles activos.

#### 5.2.1.1 Arquitectura del sistema

El objetivo de la plataforma era crear un sistema unificado para administrar una red de sensores móviles. El sistema debería proveer herramientas para monitorear el estado de los sensores, definir las tareas que pueden ejecutar y proveer almacenamiento para la información descubierta. La plataforma también debería proveer métodos para navegar, filtrar y analizar información recolectada por los sensores.

El sistema se basa en la base de datos CouchDB, y todos los requerimientos cruciales de la plataforma, como almacenamiento de información y comunicación, son soportados por la base de datos. Las operaciones exitosas dependen más que nada en un despliegue y una configuración adecuada del sistema CouchDB.

Las características únicas de CouchDB, como almacenamiento orientado a documentos, replicación integrada e interfaces HTTP, fueron, según los autores, las principales razones por las cuáles se eligió. En la plataforma utiliza para:

- Proveer GUI utilizando un servidor Web integrado.
- Administrar sensores y su configuración
- Almacenar tareas y distribuirlas a través de los sensores
- Recolectar información descubierta por los sensores

Cada sensor es una aplicación simple controlando un sensor o grupo de sensores físicos. Adquiere la información del hardware, realiza un análisis específico y le envía la información procesada a la base de datos. La comunicación con la plataforma es realizada a través de requerimientos HTTP, por lo tanto, los sensores pueden ser implementados en cualquier lenguaje de programación que soporte estos requerimientos HTTP. Esta flexibilidad permite construir aplicaciones sensores en diversos dispositivos como PCs, smartphones (corriendo Android/iOS/WP) y otros dispositivos con arquitectura ARM.

Debido a la pobre calidad de conexión provista en diversos lugares y dispositivos por los módulos de comunicación de los sensores, existe la necesidad de almacenar la data recolectada y luego sincronizarla con la base de datos principal. La manera más simple de lograr esto es iniciar una copia local de CouchDB y usarlo de la misma manera que la base de datos principal. Lo único adicional que debería configurarse es la replicación. Como resultado, se obtiene un sistema poco convencional tolerante a fallas y auto-sincronizable. Los autores testearon la replicación en una red con un 80% de pérdida de paquetes y aun así todo funcionó sin problemas; toda la información almacenada en la base de datos de un sensor fue correctamente replicada al servidor principal.

CouchDB puede ser fácilmente instalado en cualquier sistema Linux entre los que se incluyen las distribuciones ARM; de todas formas, en sistemas operativos de smartphones (iOS o Android) tiene permisos de accesos limitados. Una buena solución a este problema es TouchDB, que si bien es compatible con CouchDB, su implementación no está basada en CouchDB, es una plataforma de implementación propia. TouchDB se caracteriza por poco uso de memoria y CPU y arranca mucho más rápido que CouchBase Mobile. TouchDB en comparación con CouchDB es como SQLite con MySQL.

La plataforma requiere que cada sensor cree un documento del sensor y lo actualice cada 30 segundos para ser considerado activo. El documento del sensor contiene campos describiendo características únicas como id, tipo, estado, nombre, versión, tareas disponibles y configuración. Cuando la información está lista para ser enviada, el sensor debe crear un nuevo documento incluyendo la información identificando al sensor.

El sensor periódicamente busca nuevas tareas en la base de datos, las procesa y actualiza el documento de las tareas. Para recibir nuevas tareas, el sensor puede buscar en la base de datos remota usando vistas o incluso puede utilizar replicación para ver en su propia base de datos.

Los sensores recolectan diferentes tipos de información, incluyendo localización GPS, imágenes de la cámara y muestras de sonido. Estas últimas son almacenadas como adjuntos a los documentos.

Cabe destacar que la plataforma no fuerza ninguna estructura de datos particular, cualquier objeto JSON combinado con un adjunto binario opcional podría funcionar.

#### 5.2.1.2 Tests y resultados

Se han preparado y testeado cuatro escenarios para evaluar varios aspectos de la performance de la plataforma. Los primeros dos escenarios constaban de 5000 sensores escribiendo simultáneamente datos aleatorios en la base de datos; el tercero añadía métodos para adjuntar imágenes; mientras que el cuarto buscaba determinar el máximo número de sensores que puede manejar la plataforma.

Comparando los resultados de los cuatro escenarios se puede ver que la generación de imágenes y vistas es una tarea intensa en cuanto al uso de CPU por lo que puede ocasionar algunas bajas en la performance. En cambio, en el tercer caso el cuello de botella fue generado por la conexión de los sensores. CouchDB muestra un gran potencial almacenando los documentos binarios de los archivos adjuntados a los documentos y puede soportar la alta carga de datos del cuarto escenario. Incluso cuando había más de 12.000 sensores, el servidor se recuperó de los errores sin problemas.

La conclusión que presentan es que CouchDB parece ser una muy buena base para redes de dispositivos móviles a gran escala procesando activamente. La arquitectura del sistema es bastante simple, sin embargo, los rasgos distintivos de la aplicación y la performance son sorprendentemente buenos.

### 5.2.2 Sistema de inspecciones de edificios en construcción

Este caso trata sobre el proceso de migración de hacia una base de datos relacional a una base de datos NoSQL en una aplicación móvil para un sistema de inspecciones de edificios en construcción. Ya que la aplicación se utiliza en sitios de construcción, debe operar sin problemas incluso cuando no hay conexión a internet disponible. Hasta ahora, el enfoque utilizado consistía en extraer partes de la información de la base de datos compartidas y guardarlas en cache en el dispositivo. Sin embargo, esto trae consigo el conflicto entre datos que podrían actualizarse simultáneamente y la necesidad de actualizar el cache continuamente. El acceso a los datos entre los distintos dispositivos y el servidor ha sido hasta ahora una de las deficiencias claves del sistema. La solución propuesta fue la de utilizar bases de datos NoSQL diseñadas específicamente para estructuras distribuidas [15].

iSpect era la aplicación original desarrollada para iPad para el sistema de inspecciones de edificios en construcción. Asistía a los inspectores a realizar las inspecciones y entregar los reportes a la oficina central. La versión original usaba Microsoft SQL Server, el cual se usaba para guardar datos locales y a su vez proveía una API para toda la información que se descargaba previamente o que no necesitara ser accedida durante la inspección. La información que sí se necesitaba durante la inspección se guardaba en cache en el dispositivo mediante un archivo SQL exportado de la base de datos central. Este sistema era muy ineficiente ya que, además de ser muy incómodo, podría llevar fácilmente a inconsistencia de datos exportando archivos SQL. Más si se tiene en cuenta

que, al tratarse de una base de datos relacional, la información del sistema podría estar separada en diversas tablas, haciendo la sincronización más propensa a fallos.

#### 5.2.2.1 Arquitectura del sistema

El nuevo sistema desarrollado está compuesto por tres elementos principales que interactúan a través de la API de CouchDB: El web service CRM, una instancia CouchDB y la aplicación. Más precisamente, tanto la aplicación como el web service acceden a una base de datos CouchDB compartida **(Figura 7)**.

La aplicación a su vez está compuesta por la lógica de la misma y un DBMS local. La lógica interactúa solamente con su DBMS y nunca con el servidor CouchDB o el CRM, ya que es el DBMS local el encargado de replicar oportunamente con el servicio de CouchDB.

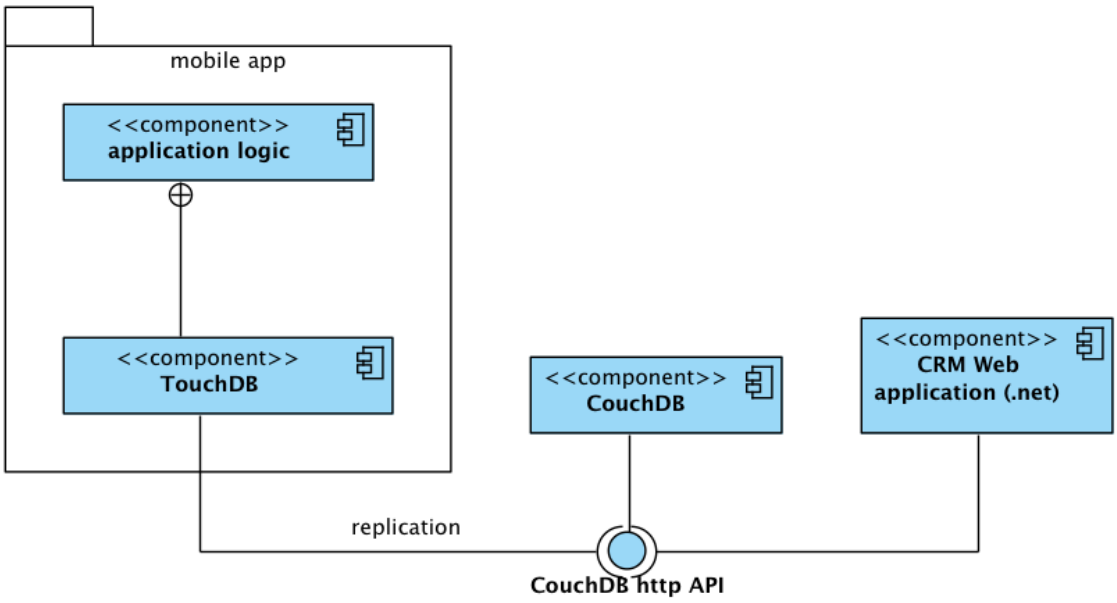


*Figura 7 Arquitectura de iSpect*

Una instancia de CouchDB en el servidor es utilizada para manejar la sincronización. CouchDB es orientada a documentos, contiene un protocolo de replicación con múltiples nodos maestros y un control de concurrencia multi-versión (MVCC) por lo que detecta y resuelve automáticamente los conflictos.

A su vez, el DBMS local de la aplicación está implementado con TouchDB. TouchDB es una base de datos embebida, más bien una librería, y un intento de portar CouchDB a dispositivos móviles. Ya que CocuhDB está escrita en Erlang y corre como un proceso independiente, no es posible instalarlo en iOS o Android debido a restricciones de las plataformas. TouchDB es escrito en

Objective-C , provee APIs que imitan las API HTTP de CouchDB y logra la compatibilidad con la misma a través de la habilidad de replicar con instancias de CouchDB. Si bien la aplicación ha sido desarrollada con TouchDB, durante el tiempo de desarrollo CouchBase Inc decidió incorporarla en su lista de productos y la renombró CouchBase Lite.

Uno de los principales puntos de la nueva aplicación era la replicación con CouchDB para aprovechar las ventajas de esta replicación, como las copias locales en el dispositivo, las operaciones de baja latencia sin retrasos por la conexión y ahorro de batería al evitar conexiones innecesarias. Se necesita que los cambios al modelo de fallas de la construcción, registrados por los diversos dispositivos, sean replicados al dispositivo desde la base de datos maestro. También se requiere que los fallos registrados en el dispositivo sean replicados a la base de datos maestro. Por lo tanto, la replicación sucede en ambos sentidos. Para esto, se configura la instancia local de TouchDB para que replique con la base de datos CouchDB en una URL HTTP específica.

#### 5.2.2.2 Tests y resultados

Se testeó intensamente la continua replicación entre la aplicación y la instancia CouchDB en el servidor. Cuando se creaba un registro para una falla de construcción en la aplicación, éste aparecía inmediatamente en la base de datos CouchDB, lo mismo sucedía cuando la aplicación sincronizaba con el servidor para traer los nuevos registros. También se testeó la replicación con una conexión a internet interrumpida y, ni bien se restauraba la conexión, todos los cambios eran sincronizados.

Otro factor que preocupó a los desarrolladores fue la performance de TouchDB comparada con Sqlite. Pero se tomaron métricas de ambas funcionando, y descubrieron que las queries de TouchDB requerían aproximadamente el mismo tiempo de ejecución que las respectivas queries Sqlite. Incluso en algunos casos TouchDB resultó más rápida que Sqlite.

En resumen, los desarrolladores lograron las metas de trabajar sin interrupciones con replicación estando incluso desconectados de la red y de trasladar el modelo de datos relacional a documentos. Sin embargo, esto trajo un costo de trabajar con nuevas tecnologías, ya sea capacitando a la gente o portando el sistema anterior.

La conclusión a la que llegan es que las tecnologías NoSQL y su enfoque distribuido representan buenas bases para desarrollar aplicaciones móviles ya que esta tecnología resuelve algunos de los principales problemas de las bases de datos distribuidas y los protocolos de replicación.

### 5.2.3 Movendos Prototype Application

Movendos Oy fue establecido a fines del 2012 como resultado de un proyecto de investigación en el Tampere University of Technology’s Biomedical Engineering department. La idea de la compañía era convertir a los asistentes de salud personales parte del día a día de la gente. La herramienta que presentan apunta a aquellos asistentes que quieran mejorar la efectividad de su labor,

haciendo que sea más fácil asignarles nuevas tareas a sus pacientes y estar en continúa comunicación con ellos [8].

La comunicación es un requerimiento crítico de la aplicación y se logra a través de un sistema de mensajes muy similar al sistema de mensajería de Facebook. Los asistentes también pueden asignar diferentes tipos de tareas a sus pacientes, monitorear su peso, llevar un recuento de los deportes realizados, etc. Además, los pacientes pueden marcar sus resultados en el registro de la aplicación, ver un historial de su actividad y controlar su progreso.

La aplicación puede ser accedida como cualquier otro sitio web desde un navegador. Esto significa que debería ser accesible usando diferentes tipos de dispositivos, como computadoras de escritorio, tablets o smartphones.

El sistema tiene dos componentes principales, el back-end y el front-end que se comunican entre sí a través de la red. El back-end está conformado por un servidor CouchDB que hace el trabajo principal de acceder a la base de datos y dar soporte al front-end al proveer la información pedida. Toda la información está almacenada en bases de datos CouchDB mientras que el CouchDB REST API es la capa entre esas bases de datos y el front-end. A su vez, el front-end está principalmente compuesto por la interfaz de usuario. Comprende todas las páginas HTML y archivos Javascript que interactúan con el back-end y el usuario.

Sin embargo, el tema de interés a estudiar consiste en el desarrollo de una aplicación Android que tome las ventajas de dichos dispositivos, como las notificaciones y la funcionalidad offline.

#### 5.2.3.1 Arquitectura del sistema

El enfoque utilizado para la aplicación Android fue la de desarrollar una aplicación que ofrezca funcionalidad y performance optimizada. Soportar cierto nivel de funcionalidad offline fue otra razón para desarrollar la aplicación.

Uno de los grandes beneficios de utilizar el paquete nativo de Android sobre la versión Web fue justamente la posibilidad de trabajar offline sin una conexión a la red. Para esto se necesita almacenar los recursos localmente en el dispositivo, para que puedan ser accedidos sin la necesidad de pedirlos a la base de datos. Esto se logró replicando la información desde el servidor principal al dispositivo.

Nuevamente se decidió implementar la base de datos local en el dispositivo utilizando TouchDB, un motor de bases de datos ligero compatible con CouchDB adecuado para aplicaciones móviles o de escritorio. TouchDB es capaz de replicar con CouchDB y tiene prácticamente el mismo REST API que éste.

El modelo de distribución entre TouchDB y CouchDB podría tomarse como un modelo de replicación maestro a esclavo, donde el servidor CouchDB actuaría como maestro mientras que TouchDB actuaría como esclavo. Aunque en este caso sería un modelo de distribución maestro a

maestro ya que las escrituras se hacen localmente en TouchDB antes de ser replicadas en el servidor maestro **(Figura 8)**.

De todas las bases de datos, sólo dos son replicadas. La base de datos común a todos, que incluye las vistas, los archivos Javascript, imágenes, etcétera, y la base de datos personal, que contiene toda la información específica del usuario.

La replicación funciona de manera diferente entre ambas. La replicación con la base de datos común funciona en un solo sentido ya que no es escrita por el usuario y solo se actualiza con actualizaciones de software o cuando un nuevo cliente es agregado al sistema. Por otro lado, la base de datos personal se actualiza constantemente por el usuario. Estos cambios son primero escritos localmente en TouchDB y luego enviados al servidor principal. A su vez, los cambios realizados desde la web o alguna otra aplicación también son replicados desde el servidor principal al dispositivo, resultando en una replicación de doble sentido entre ambos servidores.

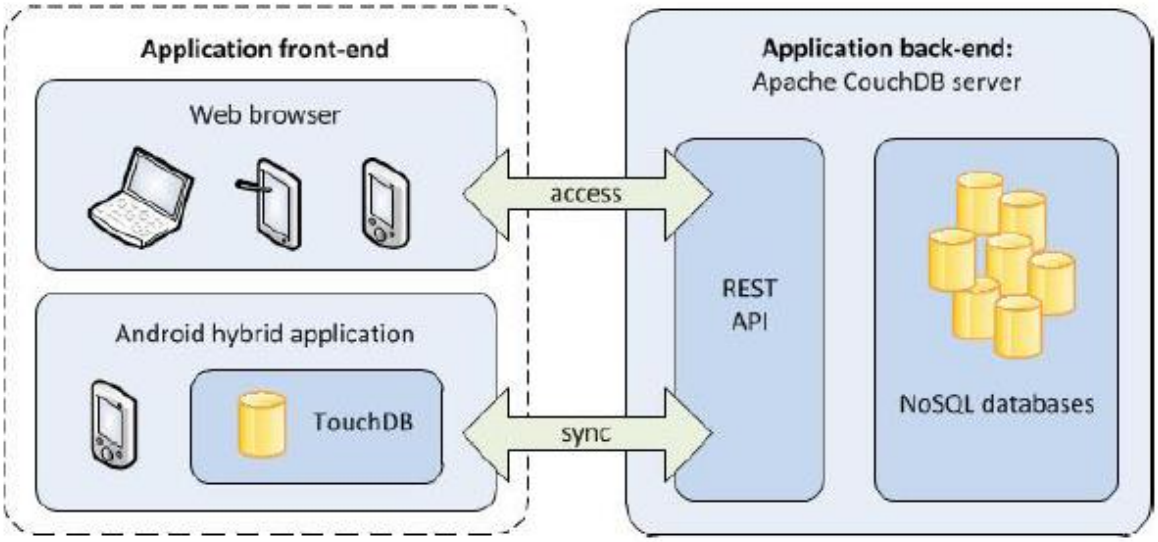


*Figura 8 Arquitectura de Movendos Oy*

De esta manera, las operaciones entre el front y el back-end siguen manejándose a nivel local, mientras el proceso de sincronización corriendo en background asegura que la información es consistente con la que se encuentra en el servidor principal de CouchDB.

Un tema importante a considerar que causó problemas con ciertos dispositivos más viejos fue la cantidad de memoria que éstos tenían. Durante la replicación, una cantidad de datos relativamente grande se transfieren y esto puede causar que ciertos dispositivos se queden sin memoria, lo que hace que la aplicación se cierre. Para resolver este problema, a replicación tuvo que ser limitada para que no todos los elementos se sincronicen sino que sólo lo hagan los que se usarían en el futuro cercano, ya sea limitando por fecha o por similitud de contenidos.

#### 5.2.3.2 Tests y resultados

La posibilidad de utilizar la aplicación de manera local y offline permitió una performance más rápida comparada con las operaciones sobre la red de datos. De todas formas, hubo otros problemas locales de performance, debido principalmente a usuarios con dispositivos más antiguos, que arruinaron en cierta medida la experiencia.

La conclusión de los desarrolladores sería que si bien se ganan ciertos aspectos con la replicación y la funcionalidad offline, el hecho de que haya que molestar a los usuarios para instalar una aplicación dedica que luego podría no funcionar cuando se tengan dispositivos más antiguos o funcionaría peor de lo que lo haría su versión web, haría replantear si la solución planteada es la mejor solución posible.

# Capítulo 6

# Lite Movie Database

Esta tesis tiene su eje central en el estudio de distintas técnicas de utilización de bases de datos NoSQL en ambientes móviles. Siendo el objetivo específico realizar un análisis comparativo de la confiabilidad de una aplicación que utilice una base de datos NoSQL con copias locales y sincronización con un servidor remoto con la de una aplicación que utilice bases de datos enteramente locales en el dispositivo.

Para lograr esto, el caso de estudio a resolver consistía en el desarrollo de una aplicación que utilice ambos enfoques de estas bases de datos NoSQL, completamente local y con replicación, y en el análisis de la utilización de ésta en diversos dispositivos en un período de tiempo. Así es como surgió Lite Movie Database [31].

## 6.1 La Aplicación

Lite Movie Database (LMD) es en pocas palabras una aplicación para Android que maneja una base de datos de películas, series y a la gente relacionada a ellas, como actores, directores, guionistas, etcétera.

Está dirigida principalmente a quiénes, por curiosidad o algún otro motivo, buscan información adicional sobre el cine o la televisión y no se quedan con lo que puedan ver en la pantalla. La idea principal es que un usuario pueda buscar dicha información de manera rápida y confiable. Ya sea en una tablet, en un celular, utilizando la red WiFi de su casa, caminando por la calle con la red móvil, o incluso sin conexión a internet; se puede acceder a la información de manera rápida y confiable.

Similar a IMDB o Wikipedia, la novedad de la aplicación radica en dos grandes diferencias: es mucho más ligera que éstas y tiene funcionalidad offline. Los artículos buscados se guardan en el dispositivo y se sincronizan con los artículos accedidos por los otros usuarios permitiendo acceder a ellos de manera rápida y segura incluso cuando no se dispone de una conexión confiable a internet.

La aplicación, como toda la información de las distintas entidades, se puede ver tanto en español como en inglés. La elección del español era obvia teniendo en cuenta el lugar desde donde se analiza, la facultad misma y que todo el trabajo de esta tesis se realizó así. El inglés, en cambio, se decidió agregar a la aplicación para que ésta pueda llegar a una mayor cantidad de personas mundialmente y que su uso sea un poco más masivo.

En la configuración, además del idioma, también se puede seleccionar cuando activar la replicación y cuando se mostrarán las imágenes para que el gasto de internet del usuario no sea mayor al deseado.

Muchos artículos contienen imágenes y el usuario puede elegir verlas siempre, verlas solamente por WiFi o no verlas nunca ya sea que no quiera gastar datos de la red móvil o simplemente no quiere verlas. La configuración por defecto es que las imágenes se vean siempre.

Algo similar sucede con la replicación. Continuamente se ejecutan muchas llamadas en background para lograr la sincronización y si estas llamadas se hacen sobre la red móvil, el consumo de datos puede ser excesivo. Por ello, el usuario puede elegir entre la sincronización por replicación sobre Wifi y redes móviles, sólo sobre Wifi o puede simplemente deshabilitarla. Por considerarlo lo correcto y para evitar que algún usuario inexperto tenga un gasto mayor de datos, la configuración por defecto es que la replicación se efectúe utilizando únicamente la red Wifi.

Otra de las características de la aplicación es que posee un historial con los últimos 100 artículos buscados o búsquedas realizadas. De esta forma, ya sea a través del menú lateral o utilizando el botón *atrás* de los dispositivos Android, se pueden rehacer las últimas acciones.

## 6.2 Funcionamiento

Cuando se ingresa a la aplicación al usuario se le pueden presentar dos pantallas, dependiendo si es la primera vez que lo hace o no. Si es la primera vez que lo hace se le presentará una foto de bienvenida y un mensaje invitando al usuario a que busque algún artículo de su interés junto con el mensaje de que hay disponible otro idioma. En caso contrario, se cargará aleatoriamente un artículo que ya haya buscado el usuario o se haya replicado del servidor **(Ilustración 1)**.

Con el motor de búsqueda del menú superior se puede consultar cualquier término y el sistema automáticamente buscará resultados, primero offline y luego online, mientras se tipea. En el cuerpo principal de la aplicación se irán mostrando los resultados.

Como se dijo, hay dos tipos de búsquedas en background que se mostrarán de la misma manera al usuario. Primero el sistema busca que el término ingresado se encuentre en la base de datos local compuesta por todo lo que ya ha buscado el usuario más lo que se ha sincronizado con el servidor y lo muestra en la lista bajo el título *Resultados Offline*. Esta búsqueda es offline y no necesita conexión a internet. En caso de no haber coincidencia, se buscará el término en FreeBase para mostrarlos en la lista bajo el título *Resultados Online*. Esta búsqueda si necesita de una conexión a internet **(Ilustración 2)**.


*Ilustración 1 Imagen Bienvenida LMD*


*Ilustración 2 Búsqueda*

Una vez que se ve la lista de resultados se puede seleccionar alguno para visualizarlo. Si el artículo seleccionado ya ha sido buscado antes y se encuentra en la lista offline se cargará y visualizará automáticamente. En cambio, si el artículo seleccionado se ve en la lista online es porque no se encuentra en la base de datos del dispositivo, ya sea porque no se ha buscado antes o no se ha sincronizado del servidor de la búsqueda de otro usuario, y deberá agregárselo. Para esto, cuando se selecciona algún ítem de la lista online se buscarán los datos de dicho ítem en *FreeBase* para luego crear el artículo y guardarlo en la base de datos local.

Una vez elegido el artículo de la lista, se visualizará. Para las películas se podrá ver: INFORMACIÓN GENERAL, DESCRIPCIÓN, PROTAGONISTAS, PRODUCCIÓN, PREMIOS y TRAILER. Para las series: INFORMACIÓN GENERAL, DESCRIPCIÓN, PROTAGONISTAS, PRODUCCIÓN, EPISODIOS, LISTA DE EPISODIOS y PREMIOS. Y por último, para las personas: DETALLES PERSONALES, BIOGRAFÍA, VIDA PERSONAL, FILMOGRAFÍA, DETRÁS DE CÁMARA: CINE, DETRÁS DE CÁMARA: TV, OTROS TRABAJOS, PREMIOS **(Ilustraciones 3 y 4)**.

Dentro de la visualización de los artículos, la información estará dividida en las secciones ya enumeradas. A su vez, el menú lateral permitirá navegar rápidamente entre estas secciones. Siendo particularmente útil en aquellos artículos que contienen demasiada información y llegar a las últimas secciones puede ser trabajoso **(Ilustración 5)**.

Algunas secciones contienen links que permiten navegar hacia otros artículos. Por ejemplo, al ver una película en la sección de protagonistas, seleccionar sobre el nombre de los actores permitirá realizar la búsqueda sobre este actor. A continuación se buscará dicha persona primero en la lista offline y luego online, y activará la secuencia del mismo modo que cuando se busca algún término en la barra de búsqueda.


*Ilustración 3 Visualización Imagen*

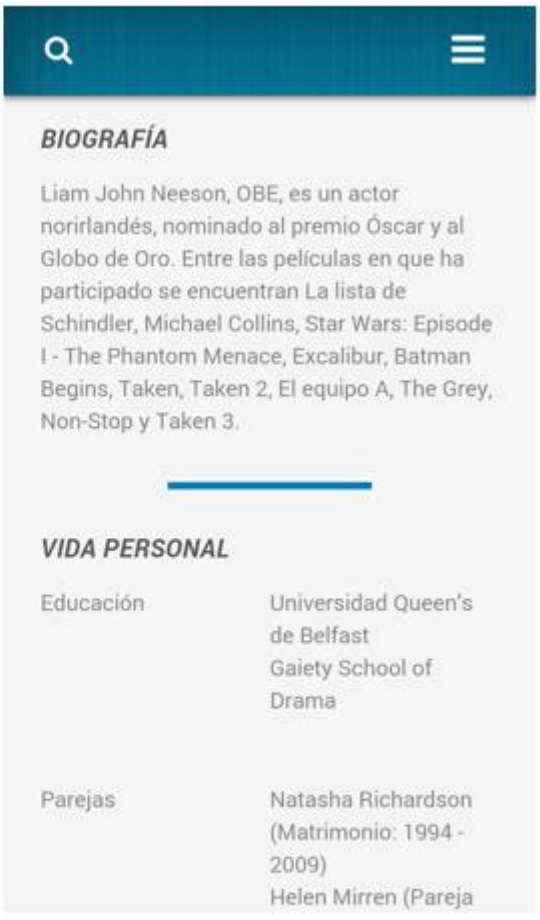


*Ilustración 4 Visualización Información*

Esta característica se puede ver en repetidas ocasiones. En las personas se puede acceder a la filmografía, a las películas producidas, dirigidas o escritas de la sección de trabajo detrás de cámara y a las series creadas o producidas en la sección de series. En las películas se puede acceder los actores de la misma, al director, a los productores y se pueden ver las películas secuelas o precuelas de la misma. En las series el mecanismo funciona de manera muy similar al de las películas **(Ilustración 6)**.


*Ilustración 5 Menú*


*Ilustración 6 Links*

Seleccionando el botón del menú de la parte superior derecha se despliega el menú de la aplicación. Como se ha dicho, el menú muestra las distintas secciones del artículo buscado y permite navegar sobre ellas. Además, permite volver a la página principal y acceder a la configuración y el historial.

En la parte de la configuración se puede configurar tres opciones claves: el idioma, la visualización de imágenes y la replicación. El idioma se puede configurar en inglés en español, traduciendo no sólo los menús, sino también los artículos. En la configuración de la visualización de imágenes se puede configurar cuándo se verán las imágenes (siempre, nunca o sólo con WiFi) para ahorrar el consumo de datos de la red móvil. Y, por último, en la configuración de la replicación se puede

configurar cuándo se sincronizarán los datos (siempre, nunca o sólo con WiFi). Esto es particularmente útil para el usuario ya que la replicación puede consumir muchos datos. Además, debajo de la configuración, se podrán ver algunas notas con respecto a la aplicación **(Ilustración 7).**

Por último, en la sección Historial se podrán ver los últimos 100 artículos vistos y búsquedas realizadas ordenadas desde el más próximo al más antiguo. Si se presiona la tecla *atrás*, que poseen casi todos los dispositivos Android, la aplicación navegará automáticamente sobre este historial también desde la entrada más próxima a la más antigua **(Ilustración 8)**.


*Ilustración 7 Configuración*


*Ilustración 8 Historial*

## 6.3 Tecnologías

PhoneGap es un framework para el desarrollo de aplicaciones móviles que permite a los programadores desarrollar dichas aplicaciones utilizando herramientas genéricas tales como JavaScript, HTML5 y CSS3. PhoneGap maneja API que permiten tener acceso a elementos como el acelerómetro, la cámara, los contactos en el dispositivo, la red, el almacenamiento, las

notificaciones, etc. Estas API se conectan al sistema operativo usando el código nativo del sistema huésped a través de una Interfaz de funciones foráneas en Javascript [21].

Las aplicaciones resultantes son híbridas, es decir que no son realmente aplicaciones nativas al dispositivo ya que el renderizado se realiza mediante vistas web y no con interfaces gráficas específicas de cada sistema. Pero no se tratan tampoco de aplicaciones web teniendo en cuenta que son aplicaciones que son empaquetadas para poder ser desplegadas en el dispositivo incluso trabajando con el API del sistema nativo.

Este tipo de aplicaciones combinan lo mejor de los enfoques web y nativos al ofrecer una aplicación web optimizada para móviles envuelta dentro de un paquete de aplicación nativo. Esta capa nativa es la responsable de proveer acceso a los componentes integrados del dispositivo. De esta forma, las aplicaciones tienen otras formas de interactuar con el usuario y su ambiente ya que los componentes mencionados pueden ser accedidos de la misma manera que se accederían con una aplicación nativa. Además, el núcleo de la aplicación sigue siendo una simple aplicación web, por lo que apuntar a diversas plataformas es más simple desde el punto de vista arquitectural [8, CAPÍTULO 1].

PouchDB es una base de datos JavaScript y open-source inspirada en Apache CouchDB y diseñada para correr dentro del navegador. Fue creada para ayudar a los desarrolladores web a construir aplicaciones que funcionen tanto offline como online ya que permite a estas almacenar datos localmente mientras están offline y luego sincronizarlo con CouchDB o servidores compatibles cuando la aplicación vuelve a estar online. Manteniendo la información de los usuarios en sincronía sin importar dónde se encuentren [22].

Ésta es una base de datos que vive dentro del navegador y la principal ventaja de esto es que no se necesita ejecutar queries sobre la red. Esto hace que las consultas sean más rápidas, ya que no son susceptibles a la red, y más confiables, ya que incluso si el usuario se encuentra offline pueden usar la información que se haya guardado localmente en cache. Incluso la información guardada puede ser información crucial para la aplicación que PouchDB puede almacenar localmente y esperará hasta que la conexión a internet sea restablecida para luego -sincronizarla con el servidor remoto [23].

PouchDB soporta todos los navegadores modernos y está completamente testeado en las nuevas versiones de Firefox (29+), Chrome (30+), Safari (5+), Internet Explorer (10+) y en Android, iOS y Windows Phone. También corre en PhoneGap, NW.js, Atom Shell y apps de Chrome. La API funciona de la misma manera en cada ambiente así se pierde menos tiempo con las diferencias de los navegadores y se puede utilizar para escribir nuevo código. PouchDB también corre en Node.js y puede ser usado como interfaz directa a servidores compatibles con CouchDB. También incluso puede correr su propio servidor compatible con CouchDB utilizando PouchDB Server.

Un tema importante a tener en cuenta, especialmente en dispositivos más viejos, es la cantidad de memoria que éstos tienen. Durante la replicación, cantidades relativamente grandes de datos son transferidas y esto puede causar que ciertos dispositivos se queden sin memoria y la aplicación

falle. Por esto mismo, PouchDB tiene diversas políticas en la cantidad de información almacenada: en algunos como Firefox y Safari el límite es alto pero preguntará al usuario si quiere almacenar más de cierta cantidad; en otros el límite es fijo como en IE y Mobile Safari; mientras que en otros como Chrome, Opera, Android y PhoneGap el límite no es fijo ni limitado, sino que es proporcional a la cantidad de memoria que tenga el dispositivo o navegador para guardar datos temporales.

Una instancia de CouchDB reservada en el servidor couchappy.com es utilizada para manejar la sincronización. CouchDB es orientada a documentos por lo que maneja la información en agregados y las vistas a través de funciones map/reduce. Contiene un protocolo de replicación con múltiples nodos maestros y un control de concurrencia multi-versión (MVCC) para manejar las diversas versiones de los documentos, guardando tanto las versiones actuales como las diversas actualizaciones que se le fueron realizando al mismo. Detecta y resuelve automáticamente los conflictos eligiendo una revisión ganadora y manteniendo las demás. CouchDB puede tomar esta decisión sin tener que negociar con otros nodos, lo que causaría retrasos.

PouchDB es una excelente opción para replicar datos desde un servidor remoto y una vez que se ha replicado la información se puede procesar y consultar. Además, CouchDB está diseñada para proveer disponibilidad a cambio de consistencia. Este modelo es perfecto para una red de nodos heterogéneos no siempre disponibles.

## 6.4 Arquitectura y desarrollo

Básicamente, la aplicación está desarrollada en Javascript, HTML y CSS y corre sobre un framework PhoneGap que le permite ser portada a dispositivos Android. El almacenamiento interno es enteramente NoSQL y está compuesto por dos bases de datos PouchDB. Sólo una de ellas sincroniza con un servidor CouchBase reservado en couchappy.com. Por último, y ya que el fin de la aplicación es analizar comportamiento, con el uso de la misma se crean registros que luego serán persistidos en una base de datos MySQL en otro servidor pero no influye con el funcionamiento de la misma.

LMD está desarrollada enteramente en HTML, Javascript (con jQuery y Angular) y CSS para darle estilos. Se puede ver en un navegador web como cualquier otra página web. Luego, utilizando Android Studio se crea un proyecto PhoneGap que emula el navegador sobre el cuál se corren los archivos. El template gráfico se ha descargado de W3layouts.

LMD contiene localmente dos instancias de PouchDB, una que se replica con el servidor y otra que no. Cuando se consulta sobre un término, primero se busca dentro de las bases de datos locales, y en caso de no encontrarlo se hace una llamada Ajax a la API de FreeBase en búsqueda de ese término. FreeBase retornará una lista de posibles resultados y cuando se elige uno se hará otra llamada Ajax que retornará toda la información de esa entidad **(Figura 9)**.

De la información retornada se filtran los valores que se consideran importantes y se los utilizan para crear objetos json. Freebase devuelve muchos datos, principalmente de uso interno como número de versión, hora de creación, usuario, etc, que se descartan para tomar solamente los valores que luego son mostrados en la app. Luego, dichos objetos se guardarán en ambas bases de datos PouchDB.

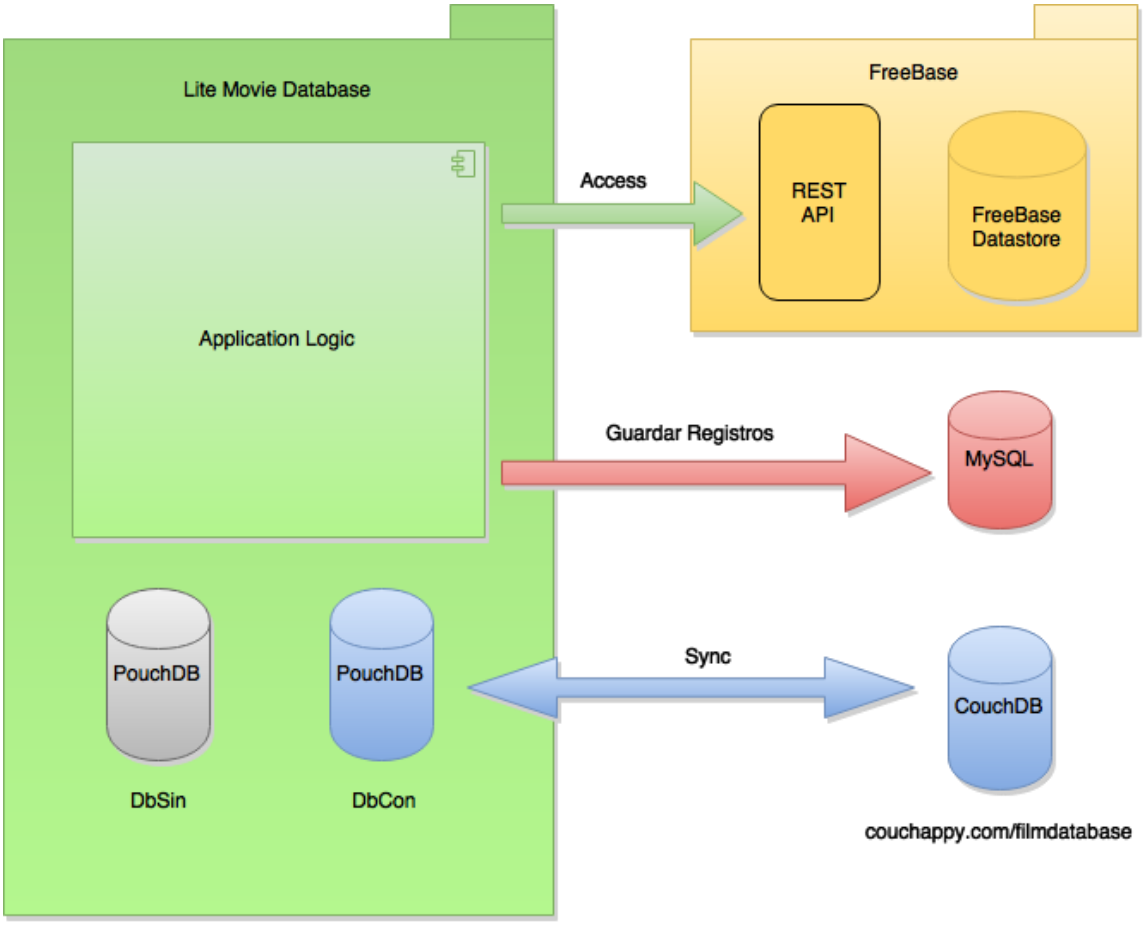


*Figura 9 Arquitectura de Lite Movie Database*

Para acceder a los datos almacenados mediante el buscador se utiliza el id de los documentos creados. Cuando un usuario ingresa un término se debe buscar entre los id de los documentos de la base de datos no sólo el término buscado sino también verificar que el idioma en el que se encuentra el documento corresponda al idioma en que esté configurado el sistema. Además, hay información más precisa que ayuda al usuario a tomar la decisión cuando la búsqueda devuelve más de un objeto sin tener que hacer otra llamada a la base de datos y que también sirve para que no se sobre escriban los documentos cuando los títulos o nombres de una película/serie/persona son iguales entre sí. Por todo esto, la regla de creación de estos id es sumamente importante.

En LMD se decidió utilizar la siguiente regla: **idioma_titulo_año_tipo**. Siendo el año de lanzamiento para las películas, el año de comienzo para las series y el año de nacimiento para las personas. Por ejemplo, para la información en español de la película "*The Matrix*" de 1999 se utilizará el id: **es-the_matrix-1999-movie**, y para el actor Liam Neeson se utilizará el id: es-**liam_neeson-1952-actor**.

De esta manera cuando comience a buscarse "The Matrix" en español el sistema buscará automáticamente un documento que empiece con **es-the_matrix** (se reemplazan los espacios en blanco por "_" y las mayúsculas por minúsculas por limitaciones de la base de datos). En el caso de que se hayan buscado previamente las secuelas de dichas películas, el sistema devolverá:

*en-the_matrix-1999-movie*

*en-the_matrix_reloaded-2003-movie*

*en-the_matrix_revolutions-2003-movie.*

En estas situaciones, cuando la búsqueda devuelve más de un resultados, resulta útil la información extra para diferenciar los documentos. Si bien los títulos de estas tres películas son diferentes y pareciera no ser de demasiada utilidad, es particularmente útil cuando se buscan nombres de películas con muchas versiones o nombres comunes de personas. Como por ejemplo, si se busca "The Three Musketeers" **(Ilustración 9)**.

En LMD cuando se crean objetos que son almacenados en la instancia PouchDB local se replicarán y sincronizarán con el servidor CouchDB de couchappy.com. De la misma manera, cuando se disponga de una conexión confiable, se descargarán del servidor CouchDB los documentos que no se encuentren en las bases de datos locales y las versiones más recientes de los objetos que ya se encuentren en la base de datos local. De esta manera, estos documentos agregados por otros usuarios se encontrarán en nuestra base de datos local como si los hubiera creado el mismo usuario. Al momento de escribir este trabajo Couchappy estaba siendo migrado al servidor SmileUpps por lo que es probable que el acceso a la primera sea dado de baja en el futuro.

Ya que la consistencia es sacrificada, las actualizaciones son propagadas de una manera que no se puede confiar en que todas las bases de datos sean idénticas en todo momento pero que si lo serán eventualmente, logrando la consistencia eventual. Ya que la base de datos FreeBase se va actualizando continuamente, esto puede suceder también en LMD cuando una consulta de un usuario sobre una entidad puede dar un resultado que crea un objeto de la base de datos. Y si luego otro usuario busca la misma entidad antes de replicarla del servidor y si esa entidad ya ha sido actualizada en FreeBase, las versiones de ambos serán distintas. Hasta que ambas se sincronicen con el servidor y el control de concurrencia del servidor decida con cuál quedarse.

Como se comentó al principio, la aplicación maneja localmente dos instancias de bases de datos PouchDB. Ambas parecieran muy similares ya que ambas guardan los documentos de los artículos creados y ambas son consultadas cuando se busca algún término. Sin embargo la gran diferencia es que una de ellas se replica con el servidor remoto de couchappy.com mientras que la otra no.

Mientras más se use la aplicación, la base de datos que se replica se irá completando con los artículos buscados por los otros usuarios, por lo que las búsquedas del mismo artículo sobre ambas bases de datos con el tiempo deberían hacer más probable un resultado positivo en una que otra. Lo que se busca es simular dos aplicaciones diferentes y analizar cuál es el rango de aciertos en la búsqueda de documentos y si influye el hecho de que una base de datos se vaya completando automáticamente según otros nodos distribuidos.


*Ilustración 9 Diferenciación de resultados*

La base de datos local no replicada se utiliza a modo estadístico para hacer la comparación con la otra base de datos. Ésta no tiene otro uso más que el de agregar nuevos documentos, al mismo tiempo que se agregan a la base de datos replicada, y el de consultar si posee los términos buscados por el usuario. Los documentos que se muestran en la aplicación son sacados siempre de

la base de datos replicada, tanto los documentos buscados por el usuario como los documentos aleatorios del inicio de la aplicación.

Ya que el almacenamiento de la aplicación se maneja solamente con instancias de bases de datos NoSQL orientadas a documentos, los diversos registros de configuración y uso interno también deben almacenarse de esta forma. Por esto, se crearon tres documentos que son almacenados en la base de datos. El primero se encarga de los datos mínimos de configuración y se abre cuando se carga el sistema; el segundo almacena los registros que luego se persistirán para analizar el comportamiento; y el último guarda el historial. Estos tres documentos se guardan en la base de datos sin replicar ya que son de uso interno y no sólo no es necesario sincronizarlos, sino que podrían generar conflictos con los otros usuarios.

Cuando se describió la aplicación se dijo que la novedad de la misma también dependía de ser más ligera que otras aplicaciones similares como IMDB o Wikipedia. Es más ligera porque sólo trae la información absolutamente necesaria cada vez que se busca algo. Por cada película, serie o persona que se busque no se carga más de 20kb o 30kb más la imagen correspondiente, la cual se puede deshabilitar o limitar al WiFi. Tampoco hay gastos extras en el tráfico de datos, y que la navegación offline, la sincronización por WiFi y la desactivación opcional de las imágenes permiten utilizar la red de datos solamente para la información vital, evitando gastos extras de la red de datos móviles.

## 6.5 Tests y resultados

### 6.5.1 Tests

Según lo expuesto en la propuesta, esta tesis tiene su eje central en el análisis de bases de datos NoSQL así como en el estudio de distintas técnicas de utilización de dichas bases de datos en ambientes móviles. Para ello se analizó el rendimiento de una aplicación móvil que utiliza dos bases de datos, siguiendo cada una un enfoque distinto, según como se especificó en el capítulo anterior. Con esto se busca aportar datos de sustento a eventuales desarrolladores que necesiten información para decidir si el uso de replicación en dispositivos móviles sería efectivamente más conveniente para sus proyectos.

La confiabilidad del software es la probabilidad de que dicho software funcione correctamente en un ambiente específico y por una cantidad de tiempo determinada. El concepto de testear la confiabilidad del software se relaciona con testear la habilidad de un software de funcionar, dadas ciertas condiciones ambientales, durante un período de tiempo particular y es de gran ayuda para descubrir muchos problemas futuros durante la fase de diseño [24].

Se utilizó para los tests la aplicación de un Software Reliability Growth Model sobre los distintos tipos y cantidad de búsquedas fallidas que tiene la aplicación en un determinado tiempo en

diversos dispositivos, siguiendo las técnicas utilizadas en "Reliability Models Applied to Smartphone Applications" [2].

Un modelo de confiabilidad de software describe el comportamiento del proceso aleatorio subyacente de las fallas en el software a través del tiempo. Y una falla es la discordancia de la información de salida del software con respecto a los requerimientos. El principio básico de cada modelo es el encajar de manera precisa los datos del fallo con una fórmula ya existente. Entonces el modelo puede ser usado para estimar la confiabilidad actual o hacer predicciones sobre el software en el futuro.

Sonia Meskini, la autora de “Reliability Models Applied to Smartphone Applications", detalla y utiliza en dicho trabajo tres modelos: el modelo Non-Homogenous Poisson Process (NHPP) - Crow-AMSAA, el modelo Musa-Basic execution time y el modelo Musa-Okumoto. Luego utiliza estos modelos sobre dos herramientas de confiabilidad de software: RGA7 de ReliaSoft y Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS).

Para nuestro trabajo, se decidió utilizar RGA de ReliaSoft en la versión 9 y el modelo NHPP para medir el número esperado de fallas en el tiempo [25]. El proceso de Poisson No Homogéneo (NHPP por sus siglas en inglés) es un proceso de tiempo continuo que consiste en contar el número de eventos y el momento en que estos ocurren en un intervalo de tiempo. Mientras que el proceso de Poisson homogéneo cuenta los eventos que ocurren con una separación constante en el tiempo, el proceso de Poisson no homogéneo cuenta los eventos que ocurren con una separación variable en el tiempo [26]. No abarca al contenido de esta tesis la explicación detallada sobre los modelos o las diferencias entre ellos, pero para más información se puede referir al trabajo de Meskini.

Para recopilar información para los tests se distribuyó la aplicación en Google PlayStore [20]. En las primeras semanas la llegaron a descargar 150 personas, de las cuáles 75 compartieron sus registros de uso y en algo más de dos meses se obtuvieron aproximadamente 22000 registros de esos mismos usuarios. Según el uso de cada usuario se iba registrando el resultado de los accesos a ambas bases de datos, la replicada y la no replicada, marcando cuando había un acierto en la búsqueda y cuando la base retornaba un error “not_found”.

Sobre esa cantidad de errores registrados, se corrieron los tests. Se analizó primero la cantidad de errores para cada una de las dos bases de datos de manera general, y luego se subdividió el análisis, según la aplicación contara con red WiFi, red móvil o funcionara de manera offline.

Para simplificar la notación, de aquí en adelante nos referiremos como **DbCon** y **DbSin** a las bases de datos con y sin replicación respectivamente y para los tests de casos independientes a la red de estas bases de datos. Además, se utilizará **DbSin WIFI**, **DbSin DATA**, **DbSin NO**, **DbCon WIFI**, **DbCon DATA** y **DbCon NO** para los tests sobre los registros de ambas bases de datos según el tipo de conexión que poseía el dispositivo al momento de registrarse la interacción con la base de datos.

El período de prueba consistió de 73 días (equivalente a 1752 horas) y abarcó desde el 11 de marzo de 2015 hasta el 22 de mayo de 2015. Se obtuvieron 22052 registros, los cuáles se pueden ver en la **tabla 1**.

*Tabla 1 Resultados obtenidos tras el período de prueba*

| | WIFI | DATA | NO | TOTALES |
|---|---|---|---|---|
| DbSin aciertos | 2453 | 394 | 289 | 3136 |
| DbSin fallas | 4509 | 433 | 2953 | 7895 |
| DbSin total | 6962 | 827 | 3242 | 11031 |
| DbCon aciertos | 3930 | 429 | 2658 | 7017 |
| DbCon fallas | 3031 | 393 | 580 | 4004 |
| DbCon total | 6961 | 822 | 3238 | 11021 |

Utilizando RGA9 de Reliasoft se procedió a realizar las pruebas. Se tomaron aquellos registros que retornaban un error “not_found” cuando intentaron acceder a la base de datos, se los agrupó en cantidad de errores de acceso ocurridos por día y se ejecutaron las pruebas en modelo Crow-AMSAA (NHPP). De cada uno de los tests se obtuvieron los parámetros necesarios para la estimación según el modelo NHPP (Beta y Lambda) junto con el número acumulado de fallas y el tiempo promedio entre fallas demostrado al fin de las 1752 horas de pruebas y una estimación de lo que sería el número acumulado de fallas y el tiempo en promedio entre éstas al final de 10000 horas de uso del sistema.

Para simplificar la notación y para estar de acorde a las palabras utilizadas por el sistema nos referiremos como **CNOF** al número acumulado de fallas (Cumulative Number Of Failures), **DMTBF** al tiempo promedio entre fallas demostrado (Demostrated Mean Time Between Failures) e **IMTBF** al tiempo promedio entre fallas instantáneo según una variable de tiempo (Instantaneous Mean Time Between Failures).

### 6.5.2 Resultados

Primero se realizaron los tests sobre ambas bases de datos (DbCon y DbSin) independientemente de la conexión de red disponible. Sobre NUM_DE_REG de **DbSin** y NUM_DE_REG de **DbCon** se obtuvieron los siguientes resultados:

*Tabla 2 Parámetros DbSin*

| Resultados | |
|---|---|
| **Parámetros** | |
| Beta | 1,2106 |
| Lambda (Hr) | 0,9350 |
| Tasa de Crecimient | -0,2106 |
| DMTBF (Hr) | 0,1833 |
| DFI | 5,4552 |

*Tabla 3 Parámetros DbCon*

| Resultados | |
|---|---|
| **Parámetros** | |
| Beta | 0,8564 |
| Lambda (Hr) | 6,6785 |
| Tasa de Crecimient | 0,1436 |
| DMTBF (Hr) | 0,5109 |
| DFI | 1,9572 |

Para DbSin al fin de las 1752 horas de pruebas se obtuvo 0,1833 Hr como tiempo promedio entre fallas (DMTBF) y 7895 como número acumulado de fallas (CNOF).


*Figura 10 DbSin DMTBF 1752 Hr*

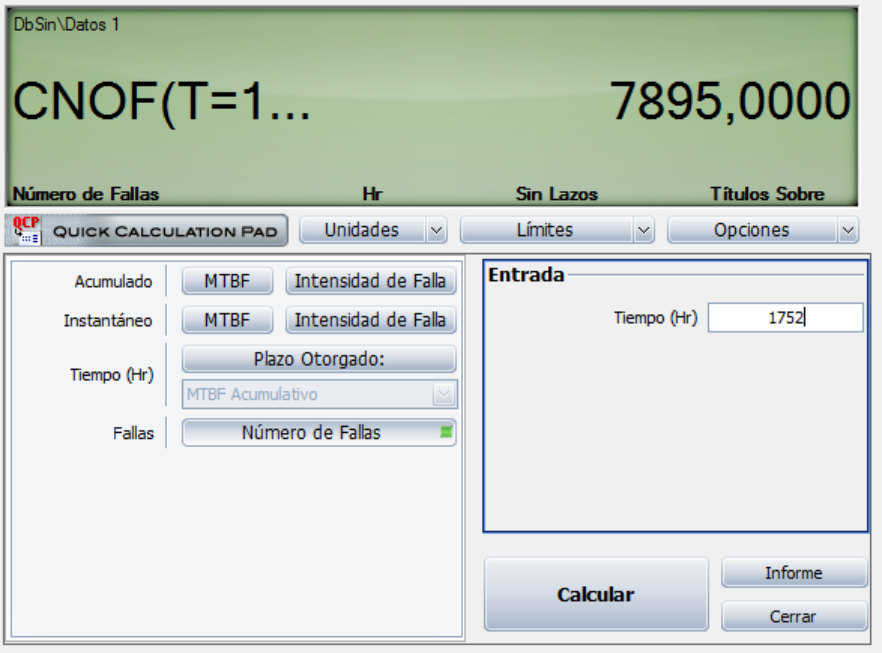


*Figura 11 DbSin CNOF 1752 Hr*

Para DbCon al fin de las 1752 horas de pruebas se obtuvo 0,5109 Hr como tiempo promedio entre fallas (DMTBF) y 4004 como número acumulado de fallas (CNOF).


*Figura 12 DbCon DMTBF 1752 Hr*

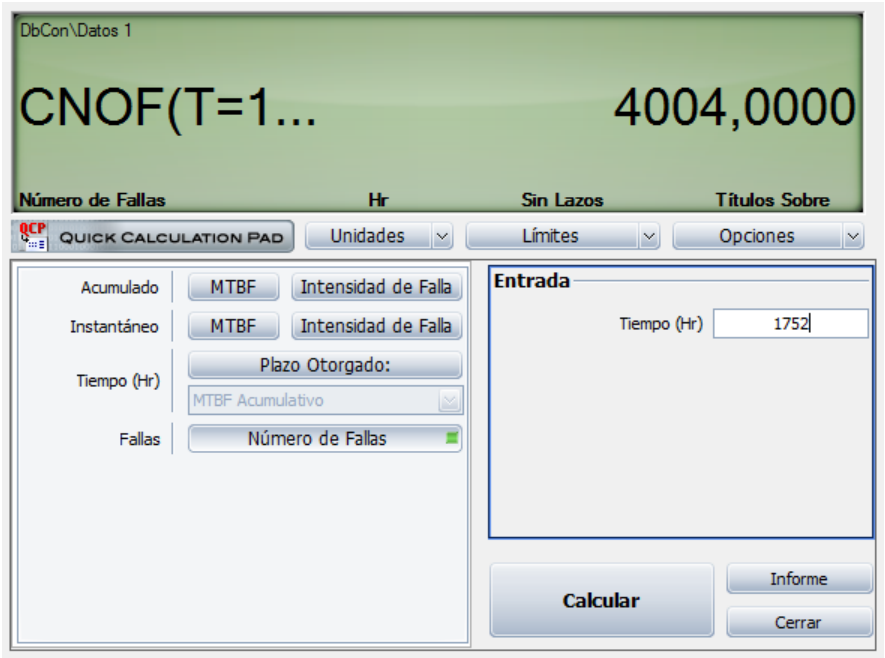


*Figura 13 DbCon CNOF 1752 Hr*

Para DbSin con una estimación de 10000 horas se obtuvo 0,1270 Hr como tiempo promedio entre fallas (IMTBF)  y 65030 como número acumulado de fallas (CNOF).

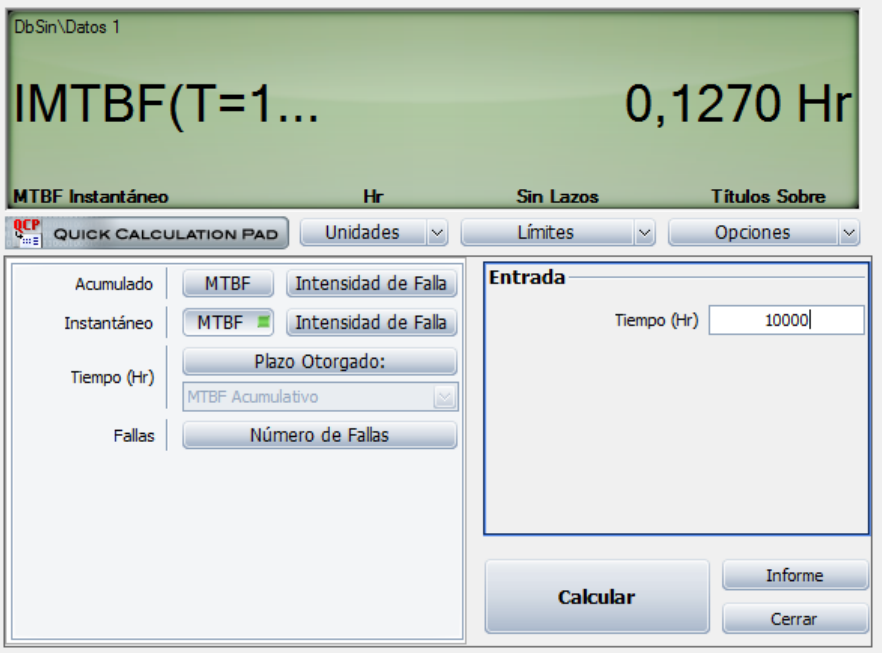


*Figura 14 DbSin DMTBF 10000 Hr*

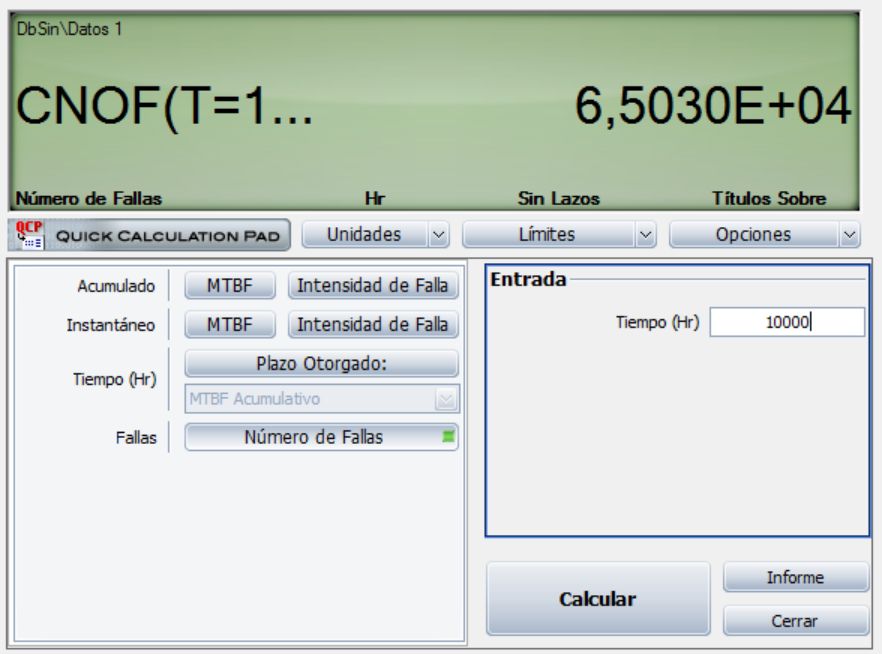


*Figura 15 DbSin CNOF 10000 Hr*

Para DbCon con una estimación de 10000 horas se obtuvo 0,6561 Hr como tiempo promedio entre fallas (IMTBF)  y 17797 como número acumulado de fallas (CNOF).

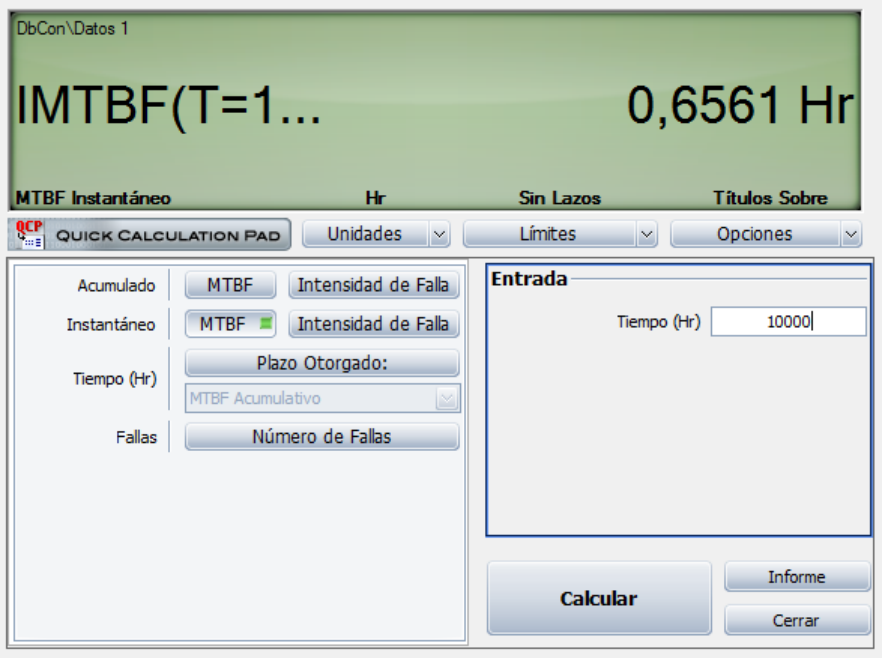


*Figura 16 DbCon DMTBF 10000 Hr*

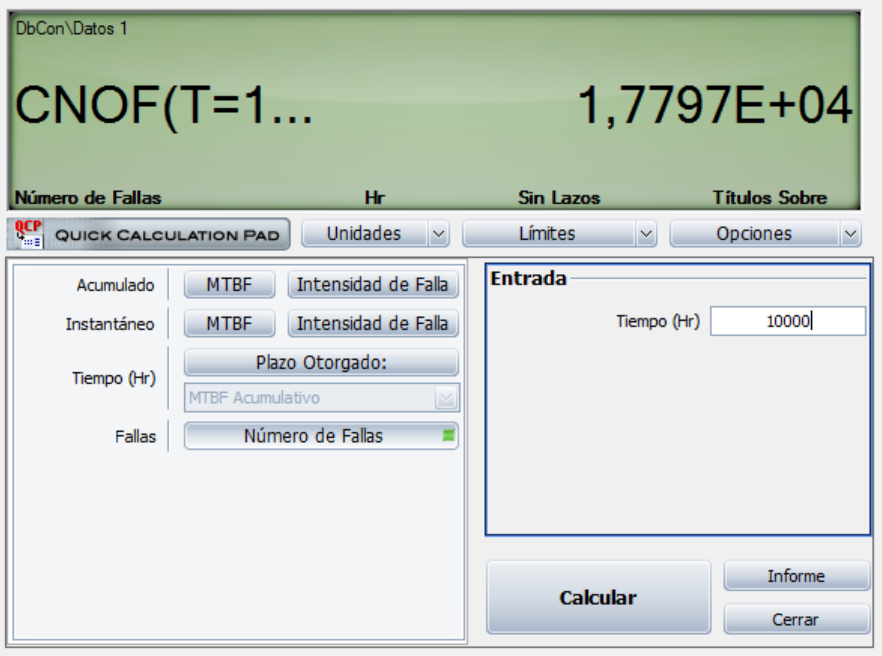


*Figura 17 DbCon CNOF 10000 Hr*

Luego se realizaron los test sobre los registros de ambas bases de datos (DbCon y DbSin) que utilizaran la red WIFI. Sobre NUM_DE_REG de **DbSin WIFI** y NUM_DE_REG de **DbCon WIFI** se obtuvieron los siguientes resultados:

*Tabla 4 Parámetros DbSin WIFI*

| Resultados | |
|---|---|
| **Parámetros** | |
| Beta | 0,8072 |
| Lambda (Hr) | 10,8597 |
| Tasa de Crecimient | 0,1928 |
| DMTBF (Hr) | 0,4813 |
| DFI | 2,0775 |

*Tabla 5 Parámetros DbCon WIFI*

| Resultados | |
|---|---|
| **Parámetros** | |
| Beta | 0,7406 |
| Lambda (Hr) | 12,0053 |
| Tasa de Crecimient | 0,2594 |
| DMTBF (Hr) | 0,7805 |
| DFI | 1,2813 |

Para DbSin WIFI al fin de las 1752 horas de pruebas se obtuvo 0,4813 Hr como tiempo promedio entre fallas (DMTBF) y 4509 como número acumulado de fallas (CNOF).


*Figura 18 DbSin WIFI DMTBF 1752 Hr*

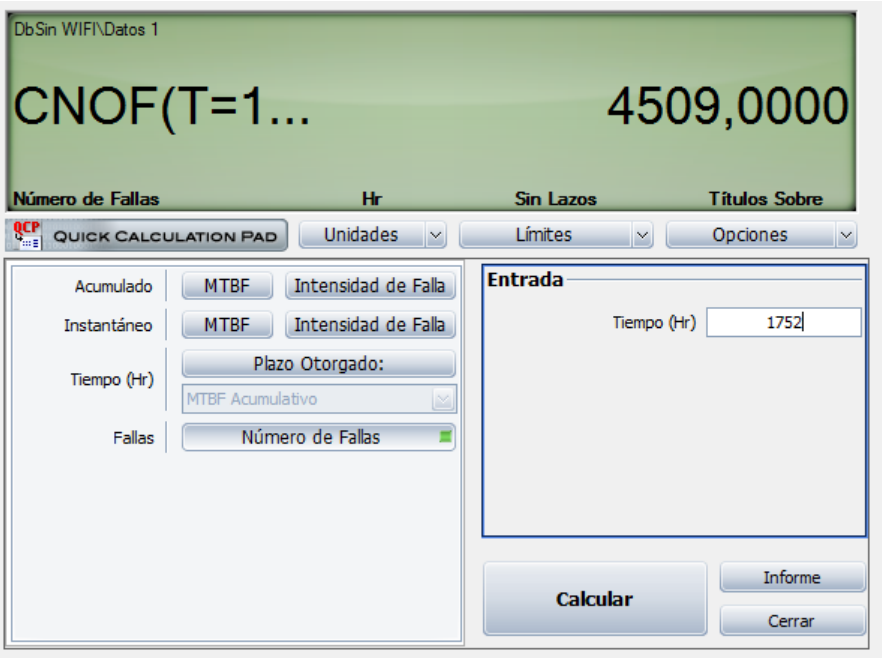


*Figura 19 DbSin WIFI CNOF 1752 Hr*

Para DbCon WIFI al fin de las 1752 horas de pruebas se obtuvo 0,7805 Hr como tiempo promedio entre fallas (DMTBF) y 3031 como número acumulado de fallas (CNOF).


*Figura 20 DbCon WIFI DMTBF 1752 Hr*

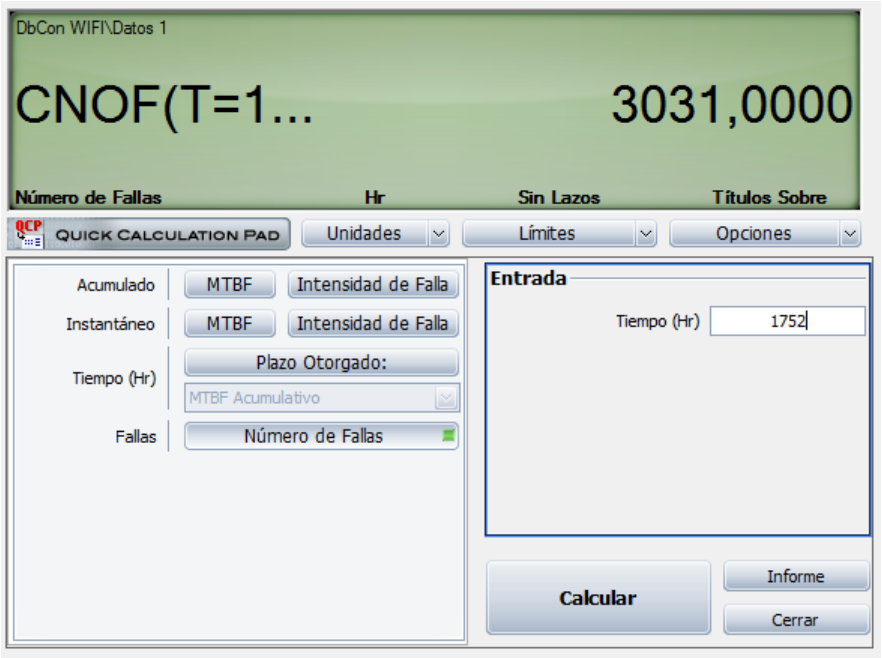


*Figura 21 DbCon WIFI CNOF 1752 Hr*

Para DbSin WIFI con una estimación de 10000 horas se obtuvo 0,6734 Hr como tiempo promedio entre fallas (IMTBF) y 18396 como número acumulado de fallas (CNOF).


*Figura 22 DbSin WIFI DMTBF 10000 Hr*

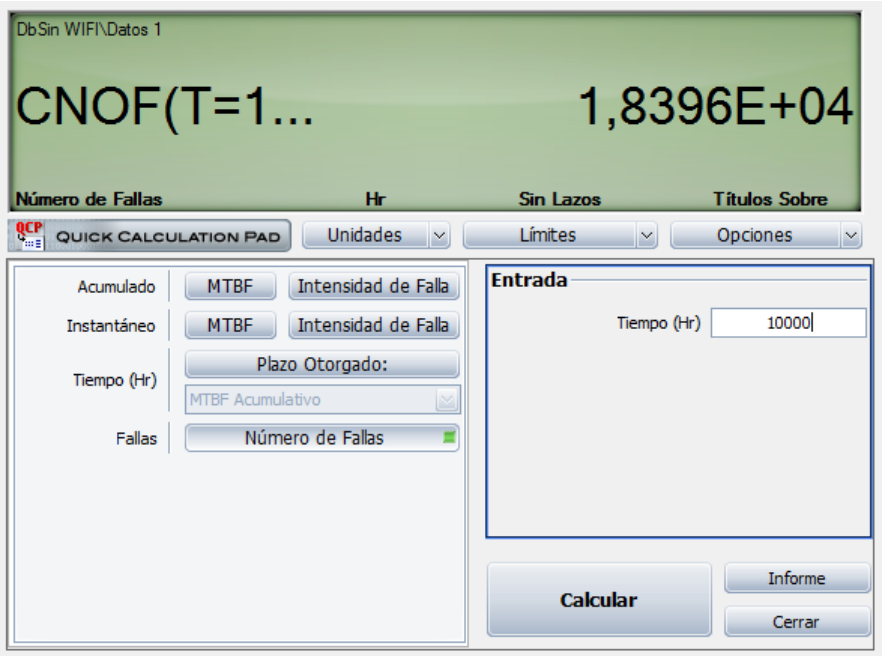


*Figura 23 DbSin WIFI CNOF 10000 Hr*

Para DbCon WIFI con una estimación de 10000 horas se obtuvo 1,2262 Hr como tiempo promedio entre fallas (IMTBF) y 11011 como número acumulado de fallas (CNOF).


*Figura 24 DbCon WIFI DMTBF 10000 Hr*

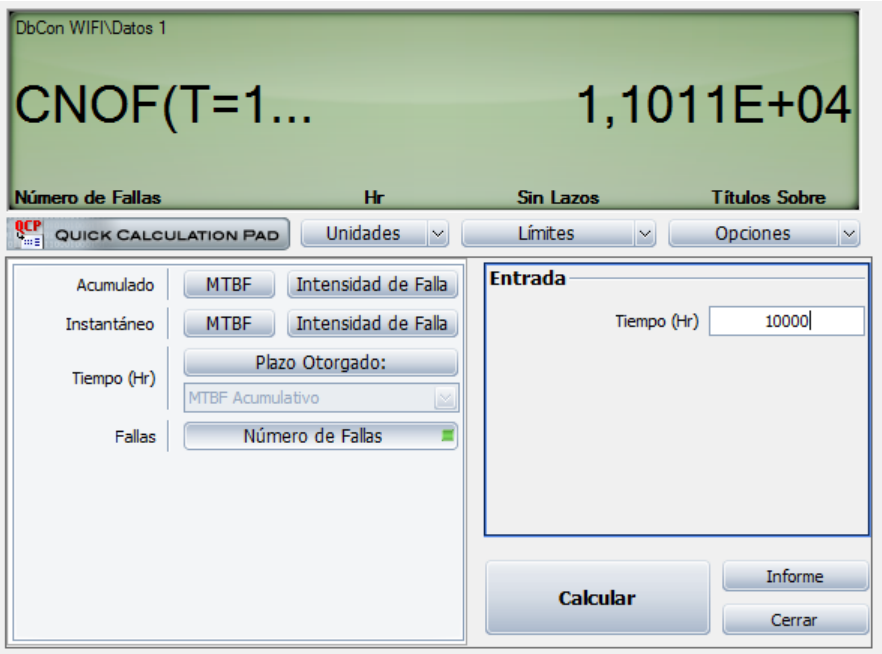


*Figura 25 DbCon WIFI CNOF 10000 Hr*

A continuación se realizaron los test sobre los registros que utilizaban la red móvil de datos de ambas bases de datos (DbCon y DbSin). Sobre NUM_DE_REG de **DbSin DATA** y NUM_DE_REG de **DbCon DATA** se obtuvieron los siguientes resultados:

*Tabla 6 Parámetros DbSin DATA*

| Resultados | |
|---|---|
| **Parámetros** | |
| Beta | 0,7993 |
| Lambda (Hr) | 1,1066 |
| Tasa de Crecimient | 0,2007 |
| DMTBF (Hr) | 5,0623 |
| DFI | 0,1975 |

*Tabla 7 Parámetros DbSin DATA*

| Resultados | |
|---|---|
| **Parámetros** | |
| Beta | 0,7684 |
| Lambda (Hr) | 1,2651 |
| Tasa de Crecimient | 0,2316 |
| DMTBF (Hr) | 5,8018 |
| DFI | 0,1724 |

Para DbSin DATA al fin de las 1752 horas de pruebas se obtuvo 5,0623 Hr como tiempo promedio entre fallas (DMTBF) y 433 como número acumulado de fallas (CNOF).


*Figura 26 DbSin DATA DMTBF 1752 Hr*

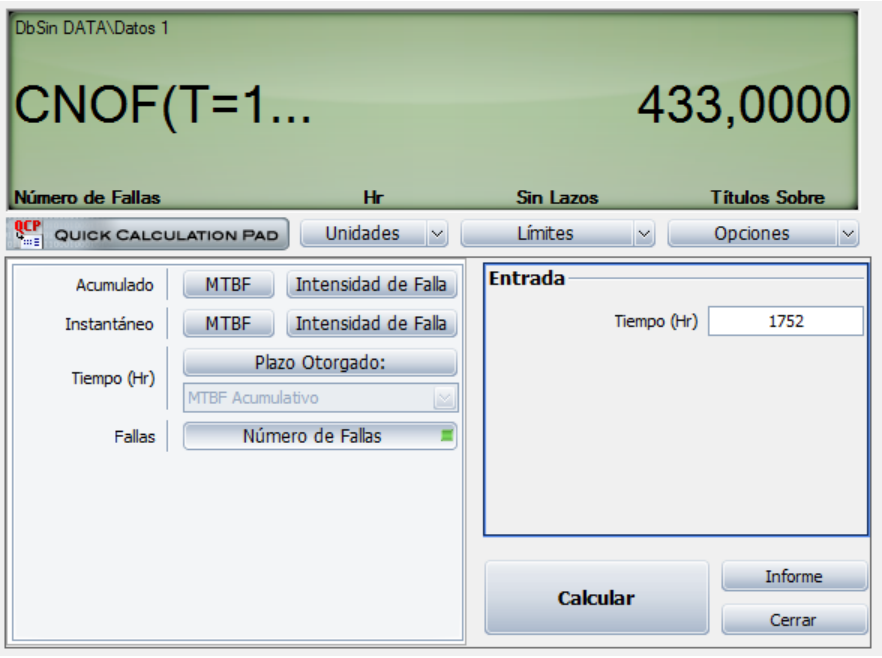


*Figura 27 DbSin DATA CNOF 1752 Hr*

Para DbCon DATA al fin de las 1752 horas de pruebas se obtuvo 5,8018 Hr como tiempo promedio entre fallas (DMTBF) y 393 como número acumulado de fallas (CNOF).

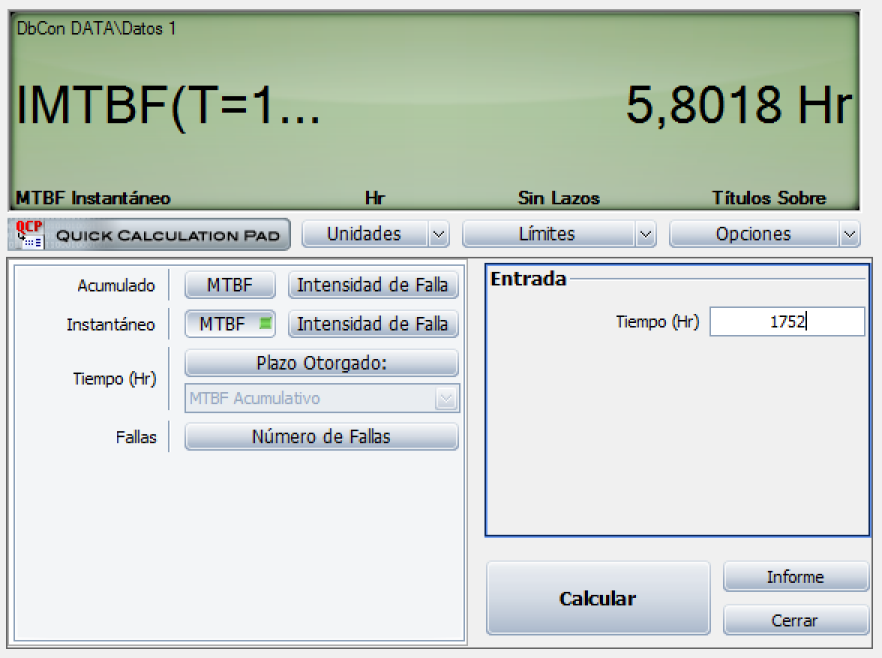


*Figura 28 DbCon DATA DMTBF 1752 Hr*

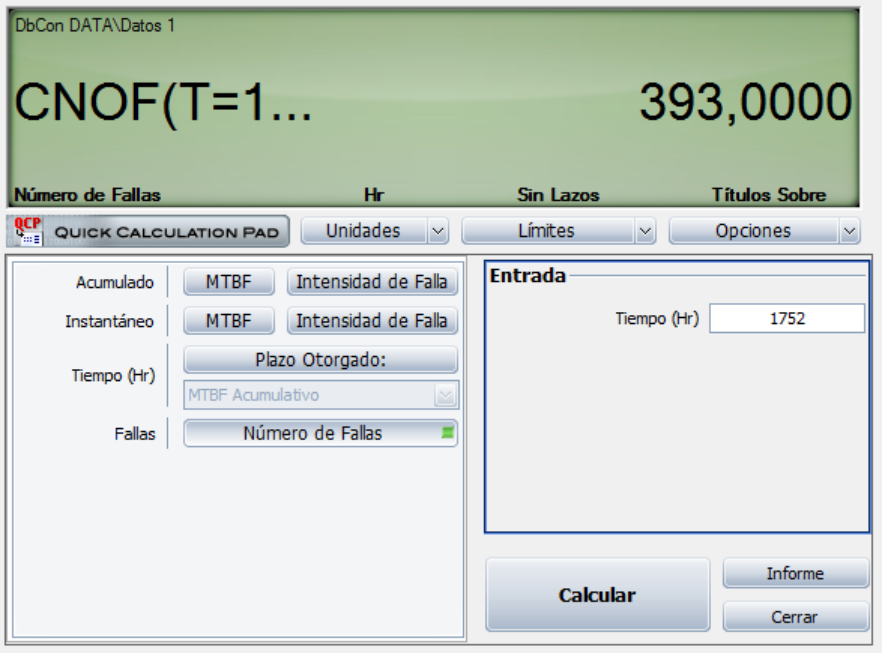


*Figura 29 DbCon DATA CNOF 1752 Hr*

Para DbSin DATA con una estimación de 10000 horas se obtuvo 7,1810 Hr como tiempo promedio entre fallas (IMTBF) y 1742,2726 como número acumulado de fallas (CNOF).


*Figura 30 DbSin DATA DMTBF 10000 Hr*

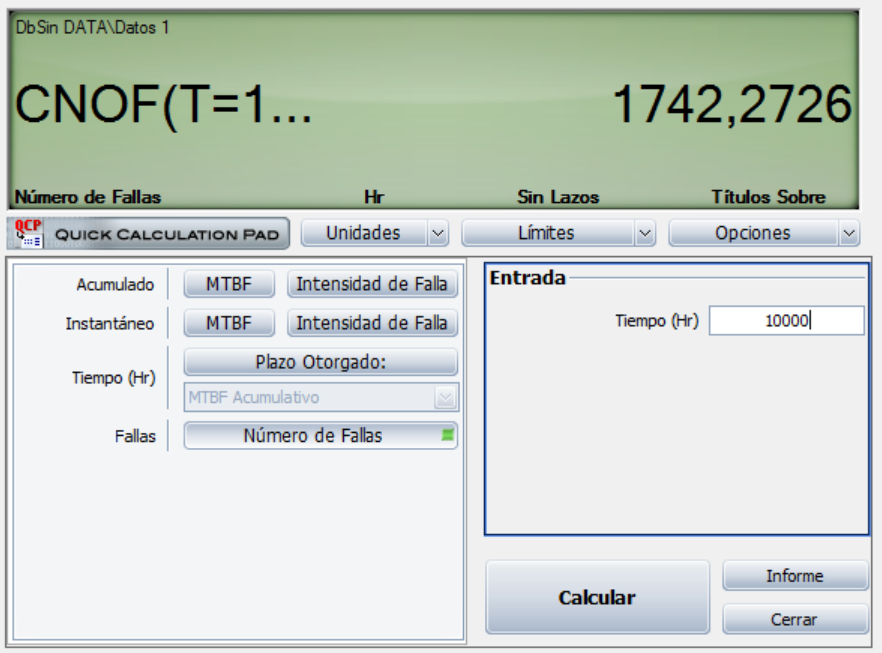


*Figura 31 DbSin DATA CNOF 10000 Hr*

Para DbCon DATA con una estimación de 10000 horas se obtuvo 8,6851 Hr como tiempo promedio entre fallas (IMTBF) y 1498,4672 como número acumulado de fallas (CNOF).


*Figura 32 DbCon DATA DMTBF 10000 Hr*

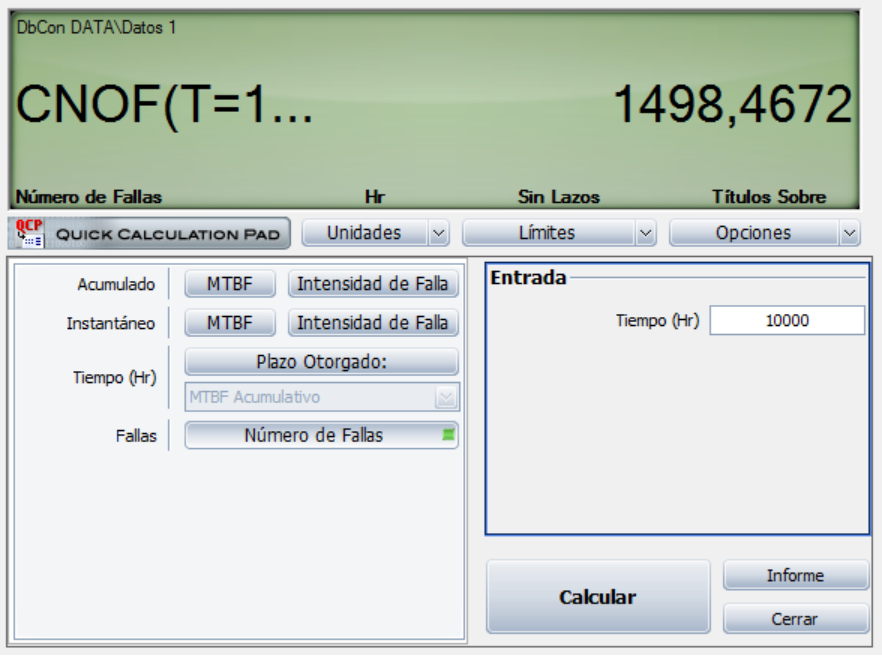


*Figura 33 DbCon DATA CNOF 10000 Hr*

Por último, se realizaron los test sobre los registros de ambas bases de datos (DbCon y DbSin) que no utilizaran ningún tipo de conexión a internet. Sobre NUM_DE_REG de **DbSin NO** y NUM_DE_REG de **DbCon NO** se obtuvieron los siguientes resultados:

*Tabla 8 Parámetros DbSin NO*

| Resultados | |
|---|---|
| **Parámetros** | |
| Beta | 7,3463 |
| Lambda (Hr) | 4,3869E-21 |
| Tasa de Crecimient | -6,3463 |
| DMTBF (Hr) | 0,0808 |
| DFI | 12,3823 |

*Tabla 9 Parámetros DbCon NO*

| Resultados | |
|---|---|
| **Parámetros** | |
| Beta | 7,9626 |
| Lambda (Hr) | 8,6410E-24 |
| Tasa de Crecimient | -6,9626 |
| DMTBF (Hr) | 0,3794 |
| DFI | 2,6360 |

Para DbSin NO al fin de las 1752 horas de pruebas se obtuvo 0,0808 Hr como tiempo promedio entre fallas (DMTBF) y 2953 como número acumulado de fallas (CNOF).


*Figura 34 DbSin NO DMTBF 1752 Hr*

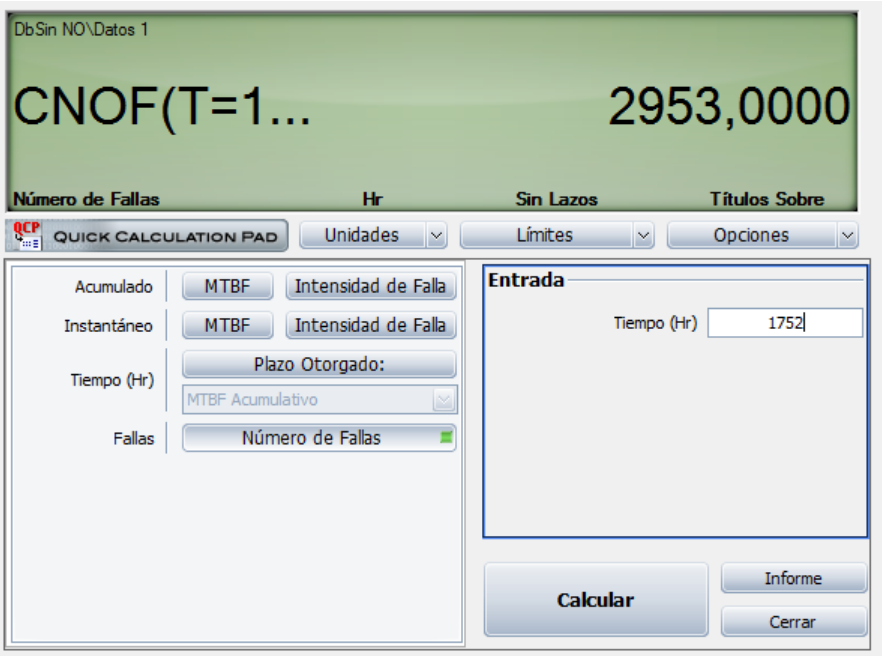


*Figura 35 DbSin NO CNOF 1752 Hr*

Para DbCon NO al fin de las 1752 horas de pruebas se obtuvo 0,3794 Hr como tiempo promedio entre fallas (DMTBF) y 580 como número acumulado de fallas (CNOF).


*Figura 36 DbCon NO DMTBF 1752 Hr*

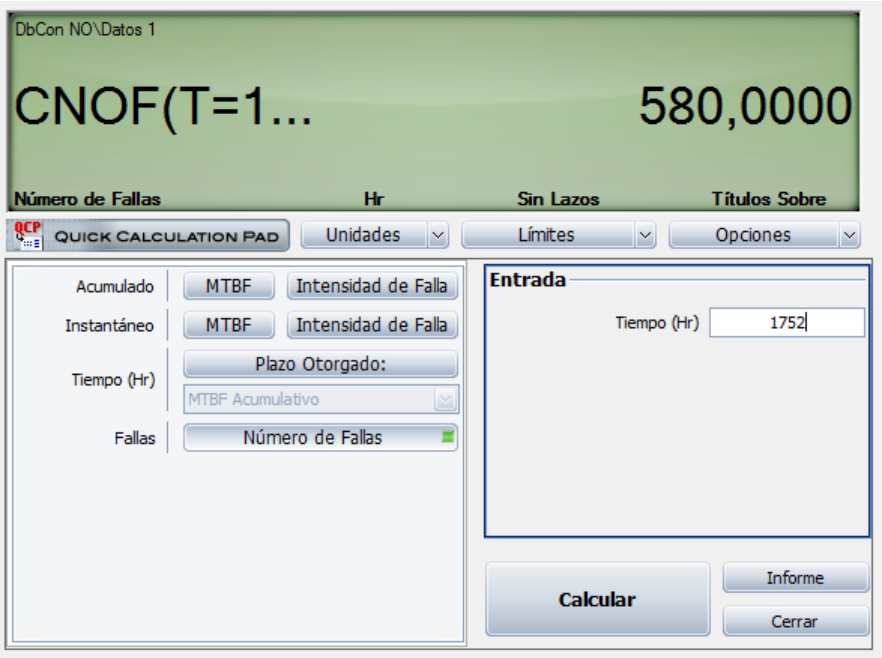


*Figura 37 DbCon NO CNOF 1752 Hr*

Para DbSin NO con una estimación de 10000 horas se obtuvo 0,0000012776 Hr como tiempo promedio entre fallas (IMTBF) y 1065400000 como número acumulado de fallas (CNOF).

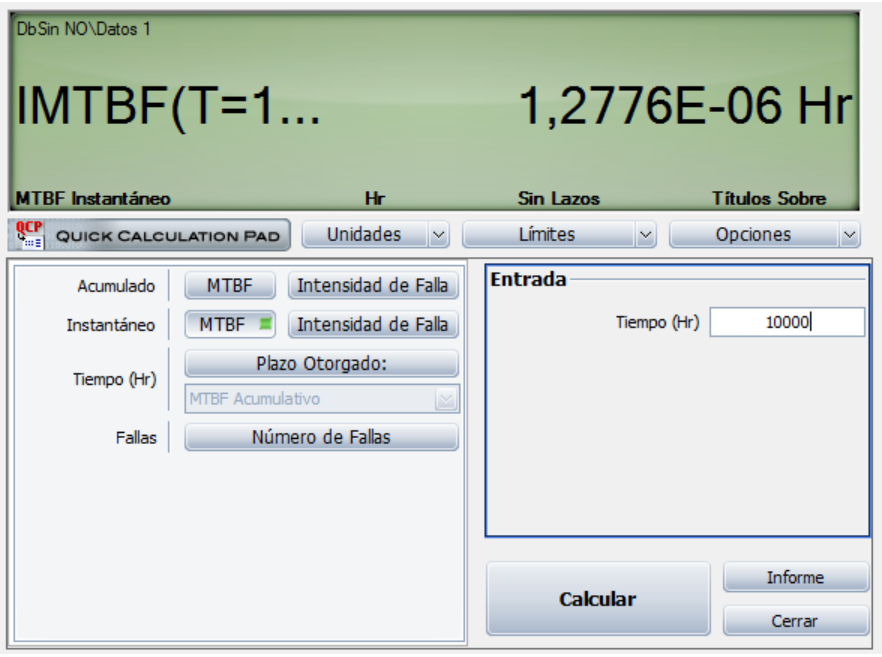


*Figura 38 DbSin NO DMTBF 10000 Hr*

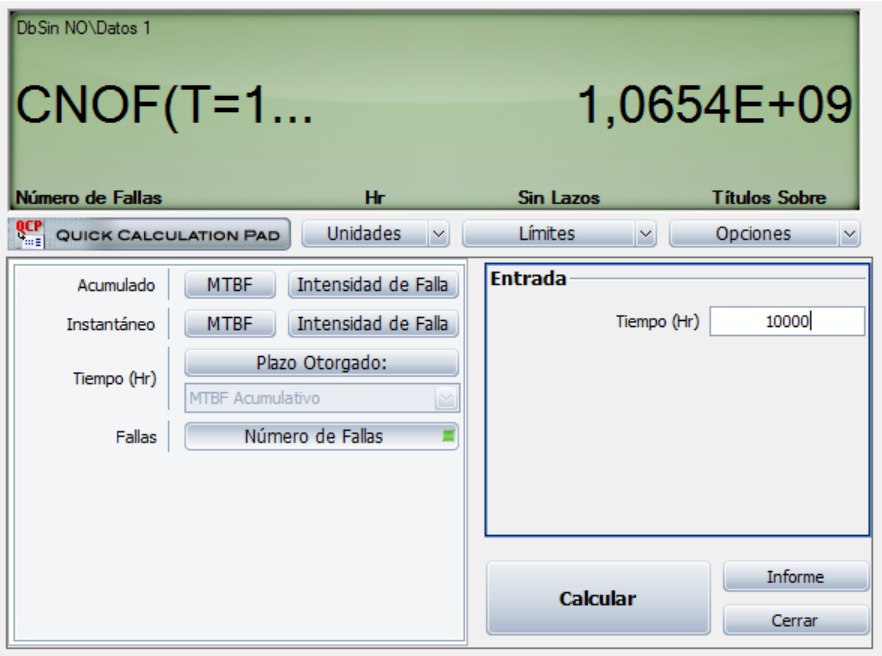


*Figura 39 DbSin NO CNOF 10000 Hr*

Para DbCon NO con una estimación de 10000 horas se obtuvo 0,0000020517 Hr como tiempo promedio entre fallas (IMTBF) y 612120000 como número acumulado de fallas (CNOF).


*Figura 40 DbCon NO DMTBF 10000 Hr*

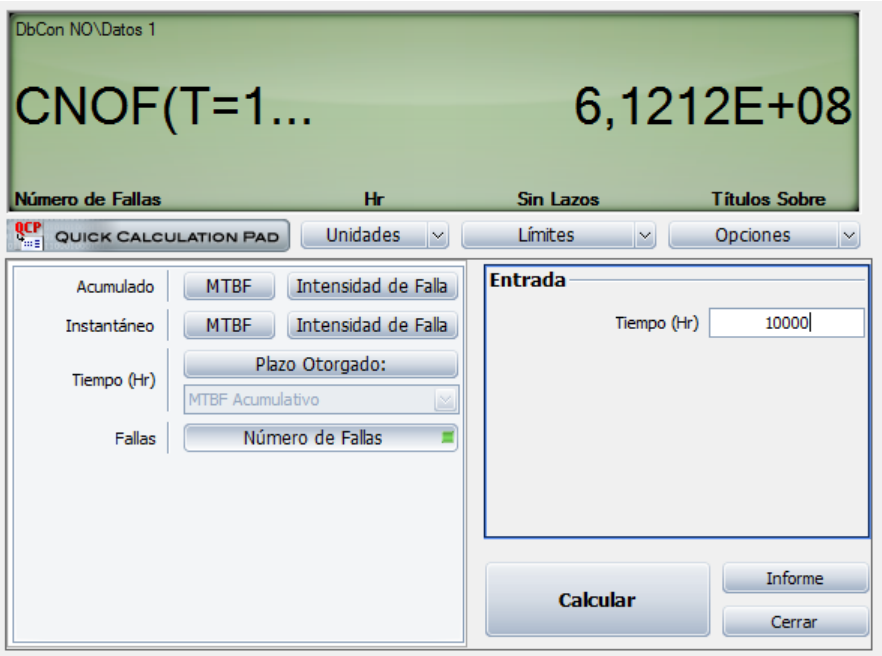


*Figura 41 DbCon NO CNOF 10000 Hr*

Podemos ver un resumen de los datos obtenidos en la **tabla 10**.

*Tabla 10 Resultados obtenidos tras las pruebas realizadas con RGA9*

| | DMTBF 1752 Hr | CNOF 1752 HR | IMTBF 10000 Hr | CNOF 10000 Hr |
|---|---|---|---|---|
| DbSin | 0,1833 Hr | 7895 | 0,1270 Hr | 65030 |
| DbCon | 0,5109 Hr | 4004 | 0,6561 Hr | 17797 |
| DbSin WIFI | 0,4813 Hr | 4509 | 0,6734 Hr | 18396 |
| DbCon WIFI | 0,7805 Hr | 3031 | 1,2262 Hr | 11011 |
| DbSin DATA | 5,0623 Hr | 433 | 7,1810 Hr | 1742 |
| DbCon DATA | 5,8018 Hr | 393 | 8,6851 Hr | 1498 |
| DbSin NO | 0,0808 Hr | 2953 | 0,0000012776 Hr | 1065400000 |
| DbCon NO | 0,3794 Hr | 580 | 0,0000020517 Hr | 612120000 |

Por lo que se puede ver en el tiempo de prueba para los datos sin injerencia de red, y bajo el mismo uso de ambas bases de datos, el tiempo promedio entre fallas para la base de datos replicada es casi el triple que el mismo tiempo para la base de datos sin replicar. Una diferencia no tan grande, pero sí bastante considerable se mantiene para el número acumulado de fallas en el tiempo de prueba. Para las estimaciones a 10000 horas de uso de la aplicación la diferencia tiende a ser aún mayor. La diferencia entre ambos es más del triple tanto en MBTF como en CNOF.

Las diferencias entre los MBTF y CNOF para los registros exclusivos al momento en que el dispositivo disponía de red WiFi no son tan grandes como cuando se hace caso omiso a la red del dispositivo. Aun así el MBTF para DbCon en el período de prueba es un 70% más grande que el

mismo valor para DbSin, e incluso el CNOF es casi la mitad del mismo valor para DbSin. Cuando se hace la estimación a 10000 horas las diferencias entre los valores de ambas bases de datos se agrandan, pero no considerablemente.

Lamentablemente, el uso de la aplicación bajo las redes móviles fue el que menos so registró. De todas maneras, se puede ver que el MBTF para DbCon es ligeramente más alto que para DbSin en el tiempo de muestra, y éste a su vez aumenta en la estimación a 10000. La diferencia entre ambos mantiene más o menos la misma relación para el CNOF.

Para los registros obtenidos cuando el dispositivo no contaba con una conexión a la red podemos ver que las diferencias de MBTF y CNOF entre ambas bases de datos son incluso mayor a estas mismas diferencias en los registros en los que se hace caso omiso a la red del dispositivo. Para el período de muestra, el MBTF de DbCon es casi 5 veces más grande que el mismo valor de DbSin y el CNOF es casi seis veces más chico en el primero que en el segundo. Aun así, analizando la estimación a 10000 del uso de la aplicación, las diferencias entre ambos tienden a acercarse. Siendo el MBTF solamente el doble y el CNOF apenas la mitad en DbCon.

# Capítulo 7

# Conclusiones y trabajos futuros

## 7.1 Conclusiones

La misma replicación que hace que se descarguen en segundo plano los documentos ya creados por otros usuarios hace que la probabilidad de buscar y encontrar un documento en la base de datos local sea mucho mayor que en aquellas bases de datos no replicadas. De esta forma, al momento de hacer una búsqueda la cantidad de veces que se necesitará una conexión a internet fiable es mucho menor en las bases de datos replicadas por lo que, teniendo en cuenta que la conectividad constante es uno de los aspectos más complicados a considerar en los dispositivos móviles, presentan una gran ventaja.

En líneas generales, bajo los tests realizados se vio que el tiempo promedio entre fallas para una base de datos es casi tres veces mayor y que el número acumulado de fallas guarda también una relación similar. Aunque si consideramos los casos en los que no se poseía ninguna conexión fiable, quizás los casos más importantes a tener en cuenta en dispositivos móviles, podemos ver que estás diferencias tienden a agrandarse.

En estos casos, la posibilidad de descargar en segundo plano cuando se posea una conexión a internet fiable otros documentos que podríamos llegar a usar en un futuro presenta claramente sus beneficios. Esto se debe principalmente a que la base de datos que se replica tiende a manejar una mayor cantidad de documentos almacenados localmente, mientras que la base de datos no replicada debe consultar a la red continuamente cada vez que se busca un documento que no haya sido buscado previamente por el usuario.

Sin embargo, esta misma ventaja presenta otros retos a tener en cuenta al momento de desarrollar aplicaciones que utilicen replicación. Muchas veces la ausencia de una conexión a internet confiable hace que la sincronización con el servidor se demore y que los documentos que se encuentren almacenados en el dispositivo no sean la última versión de los mismos. Si bien esta ausencia de red justifica una de las principales ventajas de la replicación, hay que tener en cuenta que también representa una gran desventaja de la misma.

También hay que tener en cuenta el proceso de sincronización y el espacio de almacenamiento ocupado por el mismo. Por lo que se pudo ver, el tráfico ocasionado por la sincronización pueden llegar a ser elevados, especialmente cuando se comienza a usar la aplicación. Esto no presentaría ningún problema utilizando redes WiFi pero puede presentar grandes costos si se tiene una conexión de red móvil controlada. Algo similar sucede con el espacio de almacenamiento. Si bien el framework limita los datos almacenados, en un dispositivo donde el uso eficiente del almacenamiento sea vital, guardar información que podría llegarse a usar o no en algún momento puede ser perjudicial.

El objetivo principal de este trabajo es el de aportar datos de sustento a eventuales desarrolladores que necesiten información para decidir cuál de los diversos enfoques de los mecanismos de replicación sería más conveniente para sus proyectos. Si bien puede llegar a presentar desventajas en algunos otros aspectos ya mencionados, podemos decir entonces que la incidencia de la replicación en la búsqueda es clara y presenta ventajas al momento de hacer consultas sobre diversos documentos.

## 7.2 Trabajos futuros

Hoy en día existen las tecnologías para llegar rápidamente a millones de usuarios pero aun así distribuir una aplicación, como cualquier otro trabajo personal, requiere una difusión que no siempre coincide con los tiempos planteados. Sería interesante a futuro lograr una mayor difusión de la aplicación para lograr más usuarios, y por lo tanto más registros, para correr nuevamente los tests con mayor cantidad de información recopilada.

Esto permitiría a su vez realizar un estudio más extenso sobre el uso de las bases de datos, pudiendo incluso centralizar el análisis de manera más detallada sobre el manejo de la aplicación bajo una de las opciones de red. Incluso se podría analizar exclusivamente el uso de la aplicación de una forma completamente offline con una conexión periódica a la red y la incidencia de la frecuencia de la conexión a internet en el uso de la aplicación.

Por otro lado, también podría extenderse el desarrollo de la aplicación a otros usos y así evaluar los efectos de la replicación en otros ámbitos. Extenderse más allá de las aplicaciones de consulta de información, evaluar el uso de archivos adjuntos en los documentos NoSQL o desarrollar aplicaciones para compartir información entre un selecto grupo de usuarios podrían ser algunos usos a tener en cuenta.

Incluso el desarrollo podría extenderse no sólo al análisis de la incidencia de la replicación en las búsquedas sino también a un estudio más completo que tenga en cuenta tanto dicha incidencia como así también los posibles conflictos entre los documentos replicados desde diversas fuentes junto con los efectos del tráfico de la sincronización y el uso de almacenamiento.

# Bibliografía


1. PERRIER, T.; F. PERVAIZ. 2013. "NoSQL in a Mobile World: Benchmarking Embedded Mobile Databases". University of Washington.

2. Sonia Meskini. 2013. Reliability Models Applied to Smartphone Applications. The School of Graduate and Postdoctoral Studies. The University of Western Ontario. London, Ontario, Canada.

3. TIWARI, S. 2011. Professional NoSQL. 1st Edition. Wrox.

4. COUCHBASE. 2013. Why NoSQL? Mountain View, California.

5. GANTZ, J.; D. REINSEL. 2010. The Digital Universe Decade – Are You Ready?. IDC iView, sponsored by EMC.

6. MACLEAN, D. 2011. A Very Brief Introduction to MapReduce. CS448G, Research Topics in Interactive Data Analysis.

7. SADALAGE, P.J.; M. FOWLER. 2012. NoSQL Distilled; A Brief Guide to the Emerging World of Polyglot Persistence. 3rd Edition. Addison-Wesley.

8. PÉREZ BLANCO, C. 2013. NoSQL databases in cross-platform development. Faculty Council of Computing and Electrical Engineering. Master's Degree Programme in Information Technology. Tampere University of Technology.

9. SATYANARAYANAN, M. 1996. Fundamental Challenges in Mobile Computing. School of Computer Science. Carnegie Mellon University.

10. HOANG, T.D.; C. LEE; D. NIYATO; P. WANG. 2013. A survey of mobile cloud computing: architecture, applications, and approaches. Wireless Communications and Mobile Computing. Volume 13, Issue 18, pages 1587–1611, 25 December 2013.

11. FOATCHE, M.; D. COGEAN. 2013. NoSQL and SQL Databases for Mobile Applications. Case Study: MongoDB versus PostgreSQL. Al. I. Cuza University of Iasi, Romania.

12. NORI, A.K. 2012. Mobile and Embedded Databases. Microsoft Corporation. Redmond, WA 98052.

13. ROUSE, M. 2012. Database Replication. Tech Target. (Disponible en: http://searchsqlserver.techtarget.com/definition/database-replication. Consultado el: 1ro de Diciembre de 2014)

14. STAGE, A. 2005. Synchronization and replication in the context of mobile applications. Joint Advanced Student School (JASS). Course 6: Next-Generation User-Centered Information Management. St. Petersburg

15. LORENZEN, K. 2013. Migrating a mobile application towards a distributed database for simplified synchronisation. Institutionen för informationsteknolog, Uppsala Universitet. Suecia.

16. ROUSE, M. 2011. Synchronous Replication. Tech Target. (Disponible en: http://searchdisasterrecovery.techtarget.com/definition/synchronous-replication. Consultado el: 1ro de Diciembre de 2014)

17. ROUSE, M. 2011. Asynchronous Replication. Tech Target. (Disponible en: http://searchdatabackup.techtarget.com/definition/asynchronous-replication. Consultado el: 1ro de Diciembre de 2014)

18. NOCUŃ, L; M. NIEĆ; P. PIKUŁA; A. MAMLA; W. TUREK. 2013. Car-Finding System With Couchdb-Based Sensor Management Platform. AGH University of Science and Technology Department of Computer Science 2013. Polonia.

19. CATTELL, R. 2011. Scalable SQL and NoSQL Data Stores. NY. ACM SIGMOD Record, Volume 39 Issue 4, December 2010, pages 12-27, originally published in 2010, last revised December 2011.

20. Lite Movie Database para Android, Google Play Store. (Disponible en: https://play.google.com/store/apps/details?id=com.tesis.localfilmdatabase. Consultado el: 26 de Mayo de 2015)

21. PhoneGap Seeks to Bridge the Gap Between Mobile App Platforms. GIGAOM. (Disponible en: https://gigaom.com/2009/04/05/phonegap-seeks-to-bridge-the-gap-between-mobile-app-platforms/. Consultado el: 22 de Abril de 2015)

22. About PouchDB, PouchDB. (Disponible en: http://pouchdb.com/learn.html. Consultado el: 26 de Abril de 2015)

23. An Introduction To PouchDB, Engine Yard. (Disponible en: https://blog.engineyard.com/2014/an-introduction-to-pouchdb. Consultado el: 26 de Abril de 2015)

24. PHAM, H. 1999. Software Reliability. 1st Edition. Springler.

25. RELIASOFT. 2015. RGA Version 9 Quick Start Guide. Reliasoft Publishing. (Disponible en: http://www.synthesisplatform.net/RGA/en/QS_RGA9.pdf. Consultado el: 24 de Mayo de 2015)

26. Poisson process, Wikipedia. (Disponible en: http://en.wikipedia.org/wiki/Poisson_process. Consultado el: 26 de Mayo de 2015)